



PROYECTO DE SISTEMAS INFORMÁTICOS
CURSO 2009/2010

Implementación de la aplicación de raytracing en un entorno de procesadores y FPGAs

Autores: Ignacio Arroyo Gómez

Luis Fernando Ayuso Pérez

Pablo Escobar de la Oliva

Director: Marcos Sánchez-Élez Martín

AGRADECIMIENTOS

De Pablo: A mi novia Clara, por aguantarme conversaciones en las que no entendía ni una palabra, gracias *amore*. A mi madre y a mi abuelita, porque sé que les hará ilusión que me convierta en ingeniero. Y a mi padre, para que sepa con convencimiento, que sé más que él al menos en una materia.

De Luis: Gracias Frede por explicarme cómo funciona el sistema mientras cruzábamos el desierto Jordano, gracias Tabas por dedicarme su precioso tiempo de viernes por la tarde en intentar repararme el Rodaballo y sobretodo Gracias madre por permitirme estudiar.

De Ignacio: A Almu, por sus consejos con el VHDL. A Marcos, por el tiempo que ha dedicado conmigo y las FPGA's. Y por último a la gente de la Universidad de Castilla-La Mancha, por echarnos una mano con el Altix.

PALABRAS CLAVE

IRRADIA

FPGA

MPI

Altix

VHDL

Raytracing

RASC

Render

DummyRasc

Rasconnect

AGRADECIMIENTOS	3
PALABRAS CLAVE	5
1. RESUMEN / SUMMARY	9
2. INTRODUCCIÓN AL PROYECTO	10
2.1 La Máquina Altix	10
2.2 Capa de abstracción RASC	11
2.3 Introducción a las FPGAs	13
2.4 Fases del desarrollo del Proyecto	17
2.5 Estado del arte	18
3. FASE 1: PROCESO A IMPLEMENTAR EN ALTIX	19
3.1 Nuestras opciones	19
3.2 La elección: Ray-tracing	21
4. FASE 2: TACHYON, EL SOFTWARE PARA ADAPTAR A ALTIX	22
4.1 Opciones consideradas	22
4.2 Introducción teórica a Raytracing	23
4.3 Tachyon, descripción y arquitectura	25
5 FASE 3: ADAPTACIÓN DE TACHYON A IRRADIA	28
5.1 Makefile	28
5.2 Código fuente	30
5.3 Eclipse	30
5.4 Subversion	30
6. FASE 4: PARALELIZACIÓN EN IRRADIA	31
6.1 División del trabajo en Tachyon	31
6.2 Compartir los datos	31
6.3 Versión recursiva en Tachyon y su equivalente Iterativo en Irradia	32
7 FASE 5: IRRADIA Y LAS FPGA's	33
7.1 Diseño y algoritmo paralelizado	33
8. FASE 6: PROGRAMACION DE FPGAS	36
8.1 Introducción a VHDL	36
8.2 Algoritmo a implementar	36
9. FASE 7: GENERACION BINARIOS PARA ALTIX	40
9.1 Entorno	40
9.2 Flujo de diseño algoritmos VHD	41

9.3 RASC Algorithm FPGA Developer's Bundle	42
9.4 Compilación y síntesis	43
9.5 Problemas y soluciones	46
10. FASE 8: SIMULACIÓN POR SOFTWARE	49
10.1 DummyRASC	49
10.2 Utilización de la API	50
11. FASE 9: INTERCONEXIÓN FPGAS	52
11.1 Llamadas y módulos	52
11.2 El tratamiento de la memoria en el caso de los triángulos	53
12 FASE 10: PRUEBAS	55
12.1 Pruebas de rendimiento Software	55
12.2 Pruebas de rendimiento de FPGA's y Rasclib	58
13 CONCLUSIONES, LOGROS Y LEGADO	61
13.1 Conclusiones	61
13.2 Logros	62
13.3 Legado	64
14 PROBLEMAS Y SOLUCIONES	65
14.1 Problemas Software	65
14.2 Problemas Hardware	65
14.3 Problemas varios	66
14.4 Conclusión de los Problemas	67
15 REFERENCIAS Y BIBLIOGRAFÍA	68
16. CONTENIDO DEL CD	69

1. RESUMEN / SUMMARY

Este proyecto de Sistemas Informáticos, el proyecto IRRADIA, se planteó como un acercamiento al incremento del rendimiento en computación de algoritmos aprovechando tecnologías no comunes aún en los sistemas de propósito general. El objetivo inicial fue el encontrar una aplicación con suficiente carga computacional, susceptible de ser modificada y adaptada, para luego mejorar sus tiempos de ejecución aprovechando técnicas de distribución de la carga entre diferentes procesadores y delegando parte del cómputo en hardware reconfigurable.

El sistema utilizado fue un sistema SGI modelo Altix 450 que dispone de múltiples procesadores Intel Itanium comunicados con un área de memoria, y dos blades que contienen dos FPGAs Virtex 4 LX 200 cada uno. Un sistema de estas características es físicamente capaz de ejecutar varios procesos simultáneamente y de esta manera acelerar, gracias a la paralelización, el algoritmo deseado.

La aplicación adaptada al servidor Altix es una aplicación basada en el algoritmo de raytracing. Nuestro objetivo fue investigar sobre las diferentes opciones desarrolladas de código abierto y elegir la que más se adaptaba a nuestro entorno y al servidor. Hemos realizado un análisis profundo de la aplicación y de la adaptación a la máquina así como de las optimizaciones que hemos logrado para una paralelización de la misma implementada mediante técnicas como MPI, hilos y mediante el uso de las FPGAs que alberga el servidor.

Hemos conseguido que la adaptación de la aplicación con el servidor sea todo un éxito, implementado una interfaz de programación de aplicaciones que simula las librerías RASC para la simulación software de cualquier algoritmo destinado a ejecutarse en las FPGAs, desarrollado otra interfaz de interconexión más modular y sencilla que las librerías RASC y a su vez diseñamos una parte del código en VHDL del algoritmo raytracing.

El proyecto concluye realizando una serie de test que comprueban la mejora de rendimiento del algoritmo haciendo uso de las tecnologías descritas.

This senior year project plans to approach the computational performance increase in algorithms, making use of technologies not commonly used nowadays in general purpose systems. The objective of the project is to find an application with a large enough workload to be modified. By making use of distributed computing and delegating part of the computation in reconfigurable hardware our goal is to improve our chosen application's performance.

The system that will be used in order to accomplish this goal is a SGI Altix 450. It is supplied with two double core processors that communicate with a memory area, in a shared or distributed way. It also contains two blades with two Virtex 4 LX200 FPGAs. Thus, the system is able to run a number of processes simultaneously and because of the programming it is therefore able to speed up the algorithm.

We have chosen an application based on raytracing algorithm and we have performed a thorough analysis of it and of the adaptation to the machine. The following document describes the optimizations we have achieved through techniques such as MPI, threads, or the use of FPGAs that the server has. These techniques have been necessary in order to adapt the code C and rewrite part of it in VHDL. We have also implemented an application programming interface that simulates the RASC library for the simulation by software of any algorithm designed to run on the FPGAs.

The project concludes with the results of tests we carried out that verify the faster performance of our chosen application using this technologies.

2. INTRODUCCIÓN AL PROYECTO

El proyecto Irradia se plantea como un acercamiento al incremento del rendimiento en computación de algoritmos, aprovechando tecnologías no comunes aún en los sistemas de propósito general. El objetivo de partida es encontrar una aplicación con suficiente carga computacional, susceptible de ser modificada y adaptada, para luego mejorar sus tiempos de ejecución aprovechando técnicas de distribución de la carga entre diferentes procesadores y delegando parte del cómputo en hardware reconfigurable.

Los sistemas multiprocesadores actuales permiten la división coordinada de tareas entre varias unidades de ejecución para que puedan completarse más rápidamente. Además al ser estos sistemas inherentemente escalables, ofrecen una ventaja latente: su actualización en términos de rendimiento se consigue, fácilmente, agregando más procesadores. En dichos sistemas se deben elegir cuidadosamente las técnicas que permitan una distribución eficiente y coherente de las tareas, utilizando la mayor parte posible de los recursos disponibles.

Por otra parte el hardware reconfigurable, aquél que viene descrito mediante un lenguaje de descripción de hardware, nos permite diseñar circuitos específicos para resolver un problema mucho más rápidamente que con un procesador común. No es una tecnología nueva, pero sí lo es su integración con los recursos de una computadora normal y con sus protocolos de comunicación.

Una vez elegida la aplicación, tendremos que analizar su funcionamiento interno, poniendo especial énfasis en el núcleo del algoritmo. Posteriormente seleccionaremos los puntos a optimizar mediante el estudio detallado de técnicas de paralelización como MPI, hilos e implementación mediante FPGA.

Conscientes de las dificultades a las que nos vamos a enfrentar, dada la gran cantidad de nuevas tecnologías que tendremos que abordar, esperamos documentar una opinión fundada del funcionamiento en diferentes situaciones de dichas tecnologías.

2.1 La Máquina Altix

La familia Altix es una familia de servidores de tamaño medio, modulares y con alta capacidad de ampliación. SGI vende estas máquinas como líderes del mercado en capacidad de computación por tamaño y por consumo.



fig 2.1 Xilinx SGI Blade

Los servidores Altix se utilizan para cálculo por sus prestaciones, tanto en áreas comerciales, científicas o militares. No es raro encontrar máquinas basadas en nodos de esta familia en las listas de computadores

Entre las opciones de ampliación de esta máquina se encuentra la de añadir *blades* de software reconfigurable, estos *blades* vienen equipados con FPGAs Virtex 4 LX200, las más densas del mercado en su momento. Cada *blade* proporciona a cada FPGA su propia memoria QDR, y tienen un bus interno de 32Gb/s en pico.

La plataforma

El sistema Altix es una plataforma NUMA con procesadores Intel Itanium.

Esto conlleva varias implicaciones:

- Accesos a memoria pueden producir bloqueos cuando la dirección es de otro nodo. El uso de MPI puede facilitar la escalabilidad del sistema, abstrayendo al programador de la arquitectura distribuida.
- Puede ser complicado alcanzar una ocupación completa de las instrucciones ejecutadas en Itanium, capaz de lanzar a ejecución 6

El procesador

Los procesadores Intel Itanium tiene tecnología EPIC, esta tecnología es capaz de lanzar a ejecución hasta 6 instrucciones por ciclo, pero es difícil alcanzar este grado de paralelismo.

El modelo específico de máquina

Algunos de los procesadores IA64 de Intel tienen capacidad para la gestión hardware de múltiples hilos de ejecución, lamentablemente el modelo de Altix que usamos, al ser la gama baja, no tiene tecnología Hyper-Threading, por lo que el uso de hilos simultáneamente con otra técnica de paralelismo no proporcionaría un incremento del rendimiento significativo. Dependiendo de lo rápido que sean los accesos a memoria y del tamaño de la cache, podría ser incluso más lento usar hilos (en conjunto con MPI).

Nuestra máquina en particular

La máquina con la que trabajamos, “el rodaballo”, solo tiene dos procesadores físicos, esto significa que la máquina solo tiene instalada una hoja (*blade*) de procesadores, en la cual vienen estos junto con la memoria RAM de la que disponen, por lo tanto estos procesadores acceden a su memoria de manera directa, sin necesidad de utilizar el enlace NUMA-Link. Esto quiere decir que a pesar de las capacidades de la máquina, mientras solo tenga una hoja de procesadores, es una máquina UMA, por lo que se recomienda el uso de hilos, gestionados por el sistema operativo, y un uso cuidadoso de la memoria, antes que el uso de MPI, que solo añadiría una capa de abstracción que ralentizaría el trabajo en general.

De hecho, en las pruebas realizadas con el software original, antes de modificación alguna, demuestran que pthreads es más rápido que MPI. Y esto se agrava si se escala la carga de trabajo.

El sistema operativo

El sistema Operativo es una distribución de GNU/Linux, versión 2.6.16. SUSE Enterprise server.

2.2 Capa de abstracción RASC

La manera de conectarse con las FPGAS desde software en un servidor Altix es utilizando las librerías RASC o RASCLib. La librería provee una API para interactuar con el driver del dispositivo en el kernel y controlar el hardware RASC (FPGAS). Se compone de una serie de llamadas de entrada/salida de los periféricos.

La librería está implementada en dos capas. La capa de más bajo nivel se denomina “COP level” y permite realizar llamadas individuales a los dispositivos. La capa de alto nivel es a nivel del

algoritmo, donde es posible realizar llamadas a todas nuestras FPGAs¹ como si fueran una misma y trabajar en paralelo con el mismo algoritmo. Esto sin duda es la parte más interesante de RASC puesto que permite que la paralelización de los dispositivos sea más fácil y transparente para los programadores.

Mediante el uso de estas llamadas podemos configurar un algoritmo en uno o varios dispositivos, iniciar el algoritmo mientras continuamos nuestra ejecución software y esperar a que el algoritmo termine recogiendo los datos de salida.

Las librerías, por tanto, nos permiten mover vectores de datos entre la aplicación software y las memorias de las FPGAS, además de transferir datos a los registros en tiempo de ejecución. Los primeros test con las librerías fueron para saber el *speed-up* que conseguiríamos al ejecutar un algoritmo cualquiera varias veces sin reconfigurar de nuevo el dispositivo.

En el capítulo de Pruebas, Test Benchmarks y conclusiones se encuentra la gráfica de rendimiento de la que obtuvimos la conclusión que era más eficiente utilizar un algoritmo con un elevado número de datos estáticos y la ejecución del mismo en vez de la reconfiguración.

Llamadas a la API RASC

La API nos permite la ejecución de una serie de llamadas para la conexión con los dispositivos. Pueden utilizarse desde C o Fortran90, en nuestro caso nuestra opción era utilizar C ya que nos encontrábamos mucho más familiarizados con este lenguaje.

Todas las funciones, estructuras de datos y constantes están definidas en el fichero `rasclib.h` que se debe incluir como cabecera en nuestro código.

Las funciones han de utilizarse en un orden concreto para que la conexión se materialice con los dispositivos. Tras analizar su funcionamiento, nos dimos cuenta que solo unas pocas funciones eran necesarias para la utilización de los dispositivos con la mayoría de algoritmos. Así que implementamos una API (`rasconnect`) mas modularizada y más sencilla de adaptar a todo tipo de implementaciones².

La estructura de llamadas de `rasclib` se puede organizar de la siguiente manera:

```
- rasclib_resource_configure(alg_name, num_devices, resrv_name);  
Configura el algoritmo en la/s FPGA/s indicando el pool con nombre rerv_name y  
marcando la/s fpgas en uso en uno o varios dispositivos (num_devices).  
  
- rasclib_algorithm_open(alg_name, RASCLIB_BUFFERED_IO);  
Con esta llamada notificamos a rasclib que usaremos uno de los algoritmos del  
registro interno en modo buffer entrada/salida.  
  
- rasclib_algorithm_send(algorithm_id, "x_in", X, size);  
Esta es una llamada de entrada de valores en la que indicamos que se moverán  
datos de memoria del host a x_in asociado en el bitstream. Se puede utilizar  
esta llamada sin necesidad de reconfigurar el algoritmo siempre que no esté en  
ejecución.  
  
- rasclib_algorithm_go(algorithm_id);  
Lanza el algoritmo indicado poniendo en funcionamiento los dispositivos  
marcados en el pool correspondiente.
```

¹ Nuestra plataforma (rodaballo) viene configurada con 4 FPGAs,

² La sección 7, Fase 9: Interconexión con las FPGAS detalla esta API

```
- rasclib_algorithm_receive(algorithm_id, "y_out", Y, size);
```

Indicamos donde se moverán los datos de memoria del dispositivo a memoria del host. Estos datos debería ser la salida del algoritmo tras su ejecución

```
- rasclib_algorithm_commit(algorithm_id, NULL);
```

Se mandan todos los comandos encolados como una lista de comandos. RASC realiza una serie de llamadas internas a más bajo nivel que se encolan, con commit enviamos todas los comandos.

```
- rasclib_algorithm_wait(algorithm_id);
```

Bloquea el hilo de ejecución hasta que todos los comandos han sido ejecutados.

- Cada una de las llamadas devuelven un valor RASCLIB_SUCCESS o error para permitir la depuración de las llamadas.

2.3 Introducción a las FPGAs

Dentro de la computación electrónica podemos distinguir entre computación vía software o vía hardware. La computación mediante software es la habitual dentro de las computadoras, ya sean de uso doméstico o grandes entornos. Proporciona una gran flexibilidad para ejecutar multitud de programas de muy diferentes propósitos apoyándose en técnicas que intentan aprovechar al máximo el hardware disponible, como la computación paralela o la distribuida. Por el contrario, la computación hardware proporciona una arquitectura especialmente diseñada para ejecutar un algoritmo. Los ASIC, de inglés *Application-Specific Integrated Circuit*, están diseñados para un propósito específico, por lo tanto hacen prácticamente imposible el poder modificar mínimamente los algoritmos a ejecutar.

Una solución para intentar encontrar un equilibrio entre estas dos soluciones son las FPGA's, y más concretamente, su integración en computadoras de propósito general. Las FPGA's, del inglés *Field Programmable Array Logic*, son unos dispositivos compuestos de bloques lógicos cuyas conexiones y funcionalidad son programables mediante un lenguaje de alto nivel, el HDL. A su vez dispone de puertos de entrada salida con los que comunicarse con el entorno exterior. También es frecuente en FPGA's de cierta entidad, disponer buffers de memoria y bancos de registros con los que interactuar con la computadora a la que están asociadas. Una vez generado el layout lógico y configurada la FPGA con él, ésta es capaz de ejecutar cálculos y algoritmos a gran velocidad aprovechando el paralelismo y la especialización de los circuitos sintetizados. La penalización de tiempos en estos dispositivos se produce cuando tenemos que reconfigurarlos cargando un nuevo código. Para mitigar este efecto, han surgido FPGA's capaces de reconfigurarse parcialmente, incrementando el SpeedUp que conseguimos al usarlas.

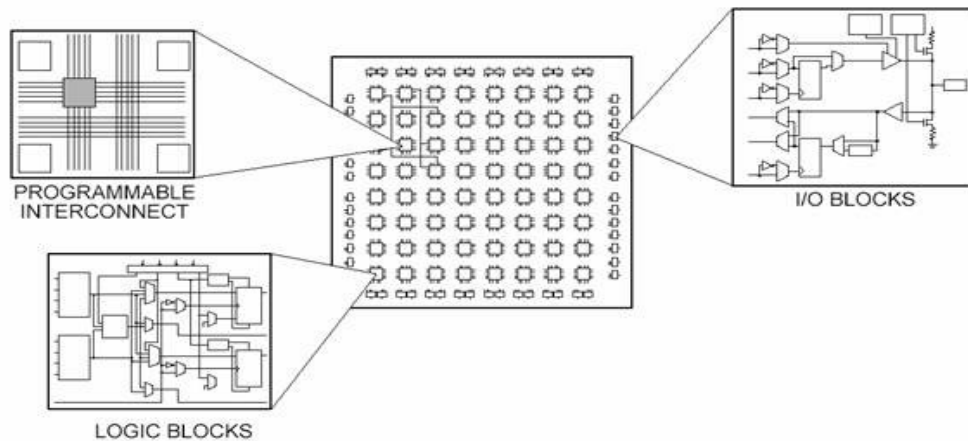


fig 2.2 Celdas en el interior de una FPGA

Familia VIRTEX 4

Para la realización de este proyecto contamos con 4 FPGA's Xilinx Virtex 4 LX200, cuyas características más importantes y que influyen directamente, como veremos más adelante, en los resultados son:

- Cada LX200 proporciona 200,448 celdas.
- 6048 Kbits de memoria RAM a 500MHz.
- 960 puertos de entrada/salida.

El identificador completo de las placas es XC4VLX200-10FF1513C, lo cual indica que nos encontramos ante el Package 1513C con un Speed=10.

En el siguiente esquema podemos observar el número de saltos que existe entre las celdas de una FPGA de la familia Virtex-4. Se puede apreciar que las conexiones más rápidas serán posibles entre celdas lógicas situadas en el centro de la placa costando como máximo tres saltos alcanzar cualquier punto del dispositivo.

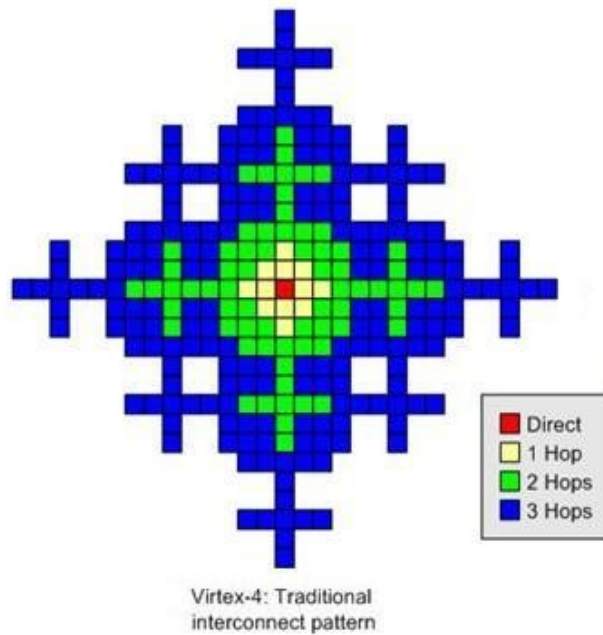


fig 2.3 Interconexión en Virtex 4

Las placas Virtex-4 LX están optimizadas para algoritmos lógicos de carácter general y ofrecían, en el momento de su lanzamiento, la mayor densidad de componentes lógicos. Las placas de esta familia integran componentes XtremeDSP, los cuales permiten implementar operaciones lentas usando técnicas de multiplexación temporal.

Entre los variados módulos que integran los XtremeDSP, los relevantes para nuestra implementación son:

- Multiplicadores complemento a 2 de 18 bits con resultado de 36 bits.
- Sumador/Restador de 48bits y 3 entradas con registro de acumulación

Como se mencionaba anteriormente, para programar la FPGA se usa un lenguaje de alto nivel, HDL, del inglés *Hardware Description Language*. Este lenguaje permite diseñar algoritmos abstrayéndonos en gran medida de los componentes electrónicos. Proporciona a su vez, los mecanismos necesarios para realizar un diseño modular y jerárquico que nos permite reutilizar o añadir componentes dentro de otros. Los dos lenguajes de este tipo más extendidos son Verilog y VHDL. En este proyecto se usa como base VHDL para la codificación de los algoritmos pero apoyado por Verilog en la parte de definición de buffers de comunicación entre las FPGA's y el computador.

Conexión con Altix

La comunicación con el computador se realiza mediante el puerto TIO ASIC que mediante unos conectores con tecnología Numalink (propia de SGI), enlaza con la memoria principal.

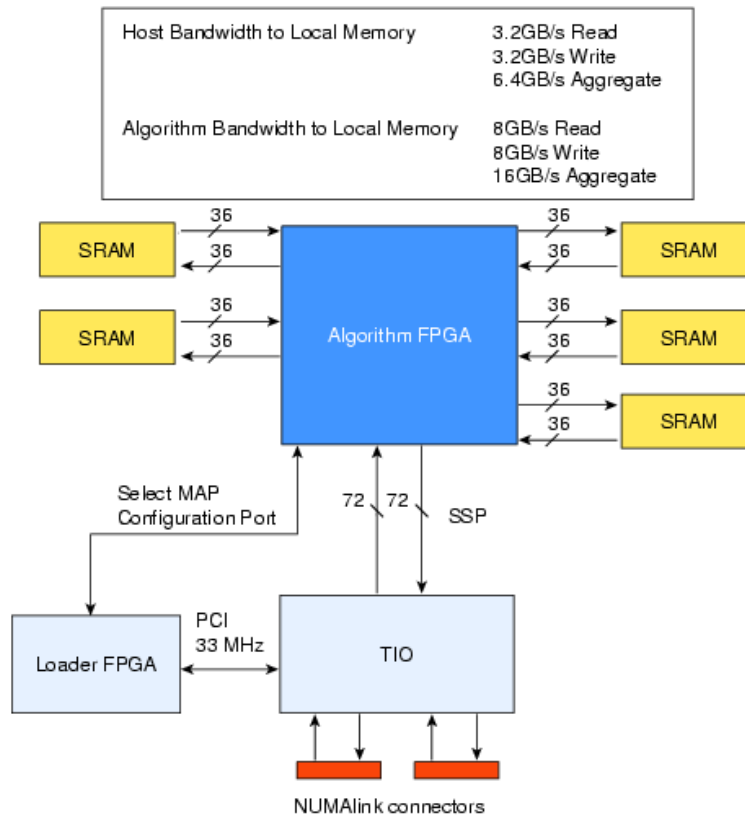


fig 2.4 Conexión Altix con FPGAs

La FPGA dispone de una memoria SRam que se puede configurar de tres maneras diferentes:

- 2 Streams de 128 bits + 1 Stream de 64 bits
- 5 Streams de 64 bits
- Sin Puertos de memoria

Además, existe la posibilidad de configurar 8 registros de propósito general y 64 de debug, todos de 64 bits. Todas estas opciones son configuradas mediante la utilidad *alg_cfg_tool* que provee RASC. Esta utilidad escribe en el fichero *alg_block_top.v* las definiciones elegidas. Todo este proceso se detalla en el apartado de generación de los .bit.

2.4 Fases del desarrollo del Proyecto

El proyecto Irradia tuvo una duración de 9 meses (de Octubre 2009 a Junio 2010). Al ser un proyecto de Sistemas Informáticos durante el curso académico surgió la necesidad de dividir el desarrollo del mismo en fases. Al ser realizado por tres compañeros podíamos dividir el trabajo de investigación por campos, y además dividirlo a lo largo del tiempo.

Las fases forman los capítulos del desarrollo del proyecto dentro de la memoria y están ordenadas cronológicamente, no obstante tuvimos que volver a algunas de ellas en las que faltaba parte de investigación o desarrollo posterior. Las primeras 2 fases tuvieron una duración de mes y medio, tiempo que teníamos planificado para ellas. Para las demás hemos dedicado la mayor parte del tiempo dividiéndonos el trabajo a su vez que lo exponíamos a los compañeros para que continuara el desarrollo de cada fase. La última fase del proyecto constituye una serie de pruebas y tests que derivan en las conclusiones de todo nuestro trabajo a lo largo del curso.

Fase 1: Elección del proceso a implementar en el servidor Altix: La primera acción a realizar con la máquina era la investigación de sus posibilidades frente a nuestras ambiciones.

Fase 2: Elección del software para adaptar en el servidor Altix: Tachyon: Introducimos el capítulo con las alternativas de desarrollos del algoritmo ray-tracing para luego realizar una introducción teórica a la aplicación que escogimos para la adaptación. Finalmente describimos la estructura de la arquitectura del software.

Fase 3: De Tachyon a Irradia: Describimos las adaptaciones que tuvimos que hacer al software Tachyon para su ejecución en la máquina Altix, el entorno de trabajo que utilizamos y cómo hicimos para la compartición del proyecto entre los compañeros

Fase 4: Multiprogramación en Irradia: En este capítulo hablamos de como realiza la carga de trabajo Irradia con las escenas 3D, además de que propuestas de optimización utilizamos posteriormente para el uso de las FPGAS.

Fase 5: Irradia y las FPGAS: Este punto describe los cambios que tuvimos que realizar en Irradia para la conexión de las FPGAs con el software, cambio de la estructura recursiva a iterativa de la programación y cambios en la distribución de los objetos de la escena para la comunicación con el algoritmo de las FPGAS.

Fase 6: La programación de las FPGAs: El código en C que implementamos en lenguaje VHDL para su ejecución paralela con el software: las intersecciones de los triángulos.

Fase 7: Generación de los binarios para Altix: Se narran los requisitos para que la generación de los binarios sea correcta, además de los añadidos que hemos realizado en el proceso de generación de éstos.

Fase 8: Simulación por software: DummyRasc: Introducción y utilización de DummyRasc: API que realizamos durante el desarrollo para poder simular las llamadas de rasclib en software. En el CD se adjunta la documentación completa de la API con su manual del desarrollador y ejemplos.

Fase 9: Interconexión de las FPGAs: Rasconnect: Introducción y utilización de Rasconnect: Otra API que implementamos durante el desarrollo para facilitar la librería rasclib y modularizarla.

Fase 10: Pruebas: Aquí se ven los resultados, las limitaciones del proyecto, el legado que dejamos.. Una serie de comparaciones en la que mostramos nuestros logros con el proyecto y el trabajo futuro que se podría continuar realizando.

2.5 Estado del arte

En la actualidad los circuitos integrados están presentes en gran cantidad de productos electrónicos industriales. Una de sus variantes, los circuitos FPGA (del inglés *Field Programmable Gate Array*) están presentes en campos tan diversos como la automoción, la electrónica de consumo, o la investigación espacial. De acuerdo al reporte “Una Guía de FPGA para Comunicaciones” de las consultoras “The Linely Group” y “iSuppli” se estima que para el pasado año fiscal 2009-2010 fue uno de los sectores de la industria de semiconductores con mayores beneficios. Generando ganancias de dos mil millones de dólares y elevando sus resultados económicos a tres mil quinientos millones de dólares para el 2013.

En el año 2010 Xilinx, la mayor empresa de investigación y desarrollo del mercado FPGA, posee en el mercado diferentes modelos de FPGA. Virtex 6: bajo consumo energético y alto rendimiento; Spartan 6: de bajo coste y bajo rendimiento; EasyPath 6: optimizadas para bajo consumo y presupuesto. En comparación con las cuatro Virtex 4 que alberga nuestro servidor Altix prácticamente doblan su rendimiento y divide a la mitad su consumo. Para las próximas generaciones la Virtex 7 continuará siendo la líder de la gama de Xilinx aumentando un cincuenta por ciento el rendimiento de la generación anterior y reduciendo un treinta y cinco por ciento el consumo.

El servidor Altix 450 forma parte de la gama de computadores de SGI: líder mundial en high performance computing (HPC). Es la compañía que los fabrica y distribuye. De los casos de éxito de Silicon Graphics en España durante el año 2010 podemos destacar dos proyectos de almacenamiento digital en red: para la empresa GolTV y para Telefónica servicios audiovisuales basados en computadores Altix 350 con un sistema de almacenamiento InfiniteStorage 4000 con capacidad para 56 Terabytes y una velocidad de 800 MB/s.

Respecto a los desarrollos específicos en servidores Altix orientados con el uso de los blades asociados con FPGA, se conocen publicaciones en el campo de la investigación, implementados a menudo con complementos como el procesador virtual Mitrion o con ROCCC un compilador optimizador de lenguaje C para FPGA. Estas investigaciones a menudo fundamentadas en cálculos científicos demuestran las capacidades del servidor unidos a las FPGA con resultados experimentales.

El algoritmo de ray-tracing, a pesar de que sus orígenes se remontan a 1968 cuando Arthur Appel ideó el algoritmo Ray Casting, hoy en día se utiliza en todo estudio de diseño 3D. El proceso de renderización puede tardar un periodo de tiempo más largo que el diseño de las escenas. Optimizado por empresas como POV-RAY o V-RAY, entre las más conocidas, se instalan como complemento o plugin de un entorno 3D para la renderización posterior de las escenas. Multimillonarias producciones de videojuegos, grandes y pequeñas empresas de visualización arquitectónica y largometrajes de animación se renderizan con esta técnica.

En Junio de 2009 Nvidia, una de las empresas punteras del mercado de procesadores gráficos, desarrolló el Nvidia Optix, motor de desarrollo diseñado para ray-tracing en tiempo real. Utiliza la arquitectura de Nvidia Cuda que permite la codificación de algoritmos para su ejecución en las GPUs. Destinado para las tarjetas gráficas de gama profesional Nvidia Quadro FX, la empresa provee el SDK (Kit de desarrollo del software) y a Junio de 2010 se encuentran en la fase beta 5 del motor. El motor también está destinado a otras disciplinas como diseño óptico y acústico, investigación de radiaciones y análisis de colisiones, en definitiva cualquier tecnología donde el algoritmo de ray-tracing se emplee.

3. FASE 1: PROCESO A IMPLEMENTAR EN ALTIX

La primera fase del proyecto fue en la que decidimos que proceso implementábamos en nuestra máquina. Una vez analizada las capacidades del servidor Altix, y el potencial de sus dispositivos asociados debíamos descartar procesos que requirieran mucho acceso a memoria secundaria.

Así pues la búsqueda la centramos en una serie de procesos que sacasen todo el partido a los procesadores Intel Itanium IA64, y a su vez la paralelización con las FPGAS Virtex4 lx200, la gama más alta de su categoría. Debíamos sacar el mayor partido de las memorias cachés compartidas por los procesadores, y de las posibilidades que brindan los dispositivos hardware reconfigurables, en concreto teniendo la hermana mayor del modelo aprovechar de todo el área hardware implementable disponible.

Existen multitud de programas de cálculo científico que requieren de mucho proceso por parte de la CPU y que minimizan los accesos a memoria (tanto primaria como secundaria). Procesos matemáticos, físicos, etc.. basados en teorías y métodos científicos. La clave se encontraba dentro de este campo de investigación.

Desde un principio nuestras preferencias eran que la máquina trabajase aprovechando lo máximo posible sus recursos, y para ello debía de hacerlo de forma paralela entre los núcleos de sus procesadores junto con sus dispositivos asociados (FPGAs). Además teníamos predilección con el lenguaje C, en parte por su facilidad con el manejo de la memoria y de los punteros, y puesto que el sistema operativo de la máquina está basado en UNIX teníamos que filtrar esa búsqueda en pro de procesos implementables en C/C++.

Asimismo nuestra ambición era que todo el cálculo realizado acabara siendo visualmente atractivo. Y no sólo eso, sino que también resultara de alguna utilidad y fuera posible reutilizarlo en otro campo.

3.1 Nuestras opciones

Nuestras opciones surgieron a raíz de todas estas reflexiones y estos estudios. Las primeras ideas que surgieron, debidas a nuestra experiencia en la rama de la ingeniería, fueron implementar algún proceso de tratamiento digital de imágenes. Dentro de este área encontramos una serie de temas específicos relacionados con lo audiovisual que nos llamaron la atención:

– Conversión de imágenes RAW.

La operativa de estos programas nos atraía por su aspecto visual, pero fué rápidamente descartada al analizar el comportamiento. Por un lado, es un proceso altamente interactivo, en el que es necesaria la toma de decisiones por parte del usuario, lo cual no lo hace apto para nuestra máquina, en la cual el proceso ideal es una carga de trabajo sin interacción ninguna. Por otra parte, cada fabricante tiene sus propios formatos RAW, por lo que se genera complejidad añadida en el código

– Video tridimensional (3D).

La falta de software especializado y documentación nos hizo evitarlo

– **Análisis de vídeo para sistemas de ayuda a la conducción.**

La necesidad de tener dispositivos o elementos de simulación complicaba en gran medida el desarrollo.

– **Reconocimiento facial en imágenes en movimiento.**

Este software nos resultó muy atractivo en un principio, con muchas aplicaciones técnicas y artísticas, pero finalmente se descartó al encontrar una objeción moral con respecto a sus posibles aplicaciones.

– **Procesos de conversión de vídeo**

Los procesos de conversión de video son procesos largos, con gran carga de trabajo y sin necesidad de interacción, por lo que se convirtieron en grandes candidatos para el proyecto, no se implementó porque nos pareció más atractivo el trazado de rayos, y porque necesitaba de demasiado acceso a memoria secundaria.

Profundizando más en el mundo del diseño y la animación 3D tuvimos que tener en cuenta que la utilización de los procesadores gráficos (GPUs) o tarjetas gráficas en estos ámbitos era mucho más eficiente que los procesadores de ámbito general. Puesto que estos procesadores gráficos están diseñados específicamente para la realización de cálculos fundamentados en operaciones vectoriales en punto flotante.

Pero existe un área en el que el rendimiento de las tarjetas gráficas no predomina sobre los procesadores y es el renderizado las imágenes. No requiere la visualización del proceso sino la visualización final. La renderización es un proceso de cálculo complejo desarrollado por un ordenador destinado a generar una imagen 2D a partir de una escena 3D. Podría decirse que en el proceso de renderización el procesador interpreta la escena en tres dimensiones y la plasma en una imagen bidimensional.

Cuando se trabaja en un programa de diseño 3D normalmente no es posible visualizar en tiempo real el acabado final deseado de una escena 3D compleja ya que esto requiere una potencia de cálculo demasiado elevada, por lo que se opta por crear el entorno 3D con una forma de visualización más simple y técnica y luego generar el lento proceso de renderización para conseguir los resultados finales deseados.

Existen multitud de procesos de renderización distintos desarrollados para los entornos de diseño 3D, durante nuestra investigación realizamos una clasificación de los mismos para poder visualizar las diferencias entre ellos. Básicamente descubrimos que se puede clasificar los procesos de *render* o *renders* en dos tipos: aquellos que se basan en las texturas de los materiales para realizar una imagen tridimensional lo más fidedigna posible a la realidad y aquellos que se basan en el tratado de la luz para el mismo fin. Nos decantamos por éste último puesto que este render trabaja por traza de rayos, y es capaz de generar reflexiones, refracciones, y sombras más precisas obteniendo un resultado visual más real.

El tiempo de los *renders* depende en gran medida de los parámetros establecidos en los materiales y luces, así como de la configuración del programa de renderizado. Normalmente cada aplicación de 3D cuenta con su propio motor de renderizado, sin embargo también existen programas o plugins desarrollados por terceros que realizan este cometido.

3.2 La elección: *Ray-tracing*

Al indagar en esta área, el algoritmo que se utiliza para la renderización de imágenes 3D basado en traza de rayos colmaba nuestras expectativas.

Habíamos llegado al proceso que respondía a todas nuestras imposiciones: máximo trabajo por parte del procesador central en detrimento de otros dispositivos del ordenador. Posibilidad de realizar parte del cálculo en los dispositivos FPGAs, puesto que el algoritmo de ray-tracing (traza de rayos) no es realmente complejo como veremos más adelante sino que repite un conjunto de operaciones con un coste computacional elevado. Posible paralelización de las operaciones. Mínimos accesos a memoria (los ficheros de escenas 3d solo enumeran coordenadas y rutas de texturas). Y por último un resultado final fotorrealista muy atractivo que puede utilizarse en muchos campos como en la animación 3D (mostrando 24 imágenes por segundo), o en diseños arquitectónicos e interiores por nombrar algún ejemplo.

Tras la investigación y gracias a que existen numerosas alternativas de *renders* basados en ray-tracing de código abierto, vimos conveniente no focalizarnos en el desarrollo de un nuevo algoritmo basado en la traza de rayos. Nuestro objetivo inicial era que el servidor Altix trabajase con un proceso de alto coste computacional, no realizar el proceso. Por ello basamos nuestro objetivo en obtener el máximo rendimiento de alguno de los procesos que ya implementados. Y la manera en la que podíamos aprovechar el potencial de la máquina Altix era utilizar la programación paralela de éstas implementaciones.

A continuación se describe la siguiente fase del proyecto en la que describimos las diferentes opciones que encontramos de implementación del algoritmo ray-tracing y la elección de uno de ellos.

4. FASE 2: TACHYON, EL SOFTWARE PARA ADAPTAR A ALTIX

Una vez tuvimos claro que usaríamos una de aplicación de Raytracing para intentar optimizar rendimientos mediante el uso de FPGA's, el siguiente paso fue buscar una que se adaptara a nuestros requisitos.

Como la idea principal del proyecto era evaluar e intentar mejorar rendimientos, debíamos buscar una aplicación escrita en C o C++ y no en Fortran, ya que no estamos familiarizados con este lenguaje. Además, estando escrita en estos lenguajes, sería fácil de adaptar o completamente compatible con la máquina Altix y su sistema operativo basado en UNIX. Otro requisito es que necesitábamos una aplicación cuyo código tuviera las licencias tales que pudiera ser modificado libremente. En un principio localizamos tres candidatos:

4.1 Opciones consideradas

The Persistence of Vision Ray Tracer (POV-Ray)

Se trata de un paquete de software muy completo precompilado para varias arquitecturas y que además provee las fuentes. Éstas están escritas en lenguaje C++.

Radiance

Es un paquete de software libre para UNIX desarrollado en el Lawrence Berkeley National Laboratory, que se orienta hacia la visualización y simulación de iluminaciones. Existen también uno equivalente para MS-DOS llamado Adeline.



fig 4.1 Radiance render

Rayshade

Es una suite libre de raytracing desarrollada inicialmente en 1988 para UNIX/X11. Usado para investigación y docencia en varias universidades del mundo.

De estos tres programas, había cosas que nos disgustaban. El principal problema de POV-Ray era que estaba escrito en C++ y nosotros buscábamos a ser posible que fuera en C por razones de rendimiento. Radiance y Rayshade nos parecieron complicados en la primera aproximación que realizamos a ellos.

Por último encontramos **Tachyon**, una herramienta desarrollada por John Edward Stone, de la Universidad de Illinois. Se trata de un programa de trazado de rayos escrito en C que ofrece un gran número de opciones de compilación dentro del propio código para ser adaptado a multitud de arquitecturas, paralelas y secuenciales. Entre las opciones de cómputo paralelo existía MPI, del inglés, *Message Passing Interface*, que fue una opción que contemplamos al inicio del proyecto y pero más tarde desestimamos al contar nuestra máquina con un sólo blade de procesadores, como se indica en el apartado de introducción al Altix. También nos gustó la multitud de configuraciones a la hora de ejecutarlo, lo que facilitaba tomar el contacto con él y realizar pruebas.

4.2 Introducción teórica a Raytracing

Una vez nos decantamos por Raytracing, como base de la aplicación elegida para ser implementada en parte sobre hardware reconfigurable, pasamos a estudiar a fondo la mencionada técnica de síntesis de imágenes tridimensionales.

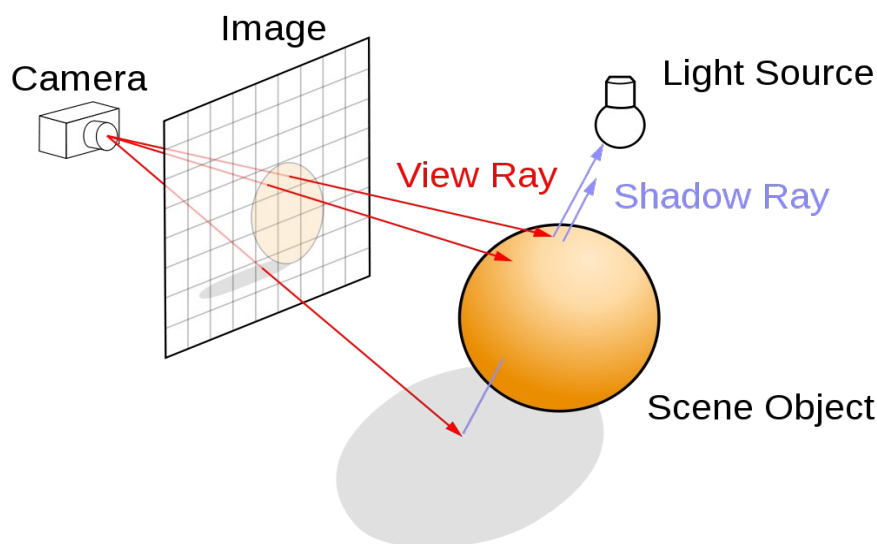


fig 4.2 esquema Raytracing

El Raytracing es una técnica muy poderosa que nos permite visualizar mundos 3D con gran calidad, llegando incluso a generar imágenes fotorealistas muy difíciles de diferenciar de la realidad. Como aspecto negativo destaca el elevado coste computacional que necesita, sobre todo si se desea una generación de *frames* en tiempo real. Actualmente es usada en grandes producciones cinematográficas distribuyendo la enorme carga de trabajo en granjas de computadores.

El algoritmo de trazado de rayos es recursivo, y consiste en lanzar un rayo por cada píxel de la pantalla y hacerlo rebotar un número determinado de veces por toda la escena. El número de rebotes determinará la calidad del acabado final.

El proceso es el siguiente: se lanza un rayo por cada píxel, y se comprueba si colisiona con cada uno de los objetos en la escena, para esta primera operación el coste es:

$O(n*p)$, siendo n el número de objetos y p el número de píxeles.

Una vez comprobados todos estos, se descartan todos aquellos píxeles que no colisionan, devolviendo para ellos el color de fondo (o el color de la correspondiente posición en la textura de fondo) con las debidas modificaciones de niebla.

Para los píxeles cuyos rayos hayan colisionado con algún objeto, se evalúa la posición en la que colisionó, se acumula el color de la posición con los debidos modificadores de luz, se computa la dirección del rayo reflejado y se hace una llamada recursiva. Después del rayo reflejado, se computa el rayo transmitido, el que atraviesa objetos no opacos, de el mismo modo. Siendo R el porcentaje de rayos reflejados en un píxel, T el porcentaje de rayos transmitidos, y n el número de objetos en la escena, el número de objetos (medio) que se chequean por píxel, en esta segunda operación, es:

$$f(x_s) = n + T/100 * f(x_{s-1}) + R/100 * f(x_{s-1}) \quad \text{siendo } s \geq 0$$

Al ser una llamada recursiva, se establece el caso base en el número máximo de saltos (s) que un rayo puede hacer, siendo por defecto 6. Este valor es el número de rebotes, que como antes indicábamos, establece la calidad del acabado final.

```
trazarImagen ()
{
    para todos los pixeles {
        rayo = rayo por los puntos (Ox,Oy) y (pixX,pixY)
        prof = 0
        trazarRayo(rayo, prof)
    }
}

trazarRayo (Rayo r, profundidad p): Color
{
    si p > limite de profundidad {retornar;}
    obj = objeto más cercano que intersecta el rayo
    rayos[] = rayos reflejados en obj
    c = (color(obj) + suma(colores_luces, intensidad_luces)) * peso(p)
    para i=1 hasta n rayos
    {
        trazarRayo(rayos[n], p+1)
    }
    retornar c
}
```


Arriba podemos ver en pseudocódigo el desarrollo de un programa que realice trazado de rayos. El algoritmo básico de Raytracing es sencillo pero posee una complejidad de $O(n^2)$. Con el elevado número de píxeles necesario para disfrutar de excelentes imágenes demandadas por el público, el coste computacional se dispara siendo necesario grandes explorar nuevas tecnologías que hagan del Raytracing una técnica asequible, tanto temporal como económicamente.

4.3 Tachyon, descripción y arquitectura

Descripción

Tachyon es un software para Raytracing en paralelo desarrollado por John E. Stone . El proyecto es muy adecuado como punto de partida para nuestro trabajo ya que es compatible con un gran número de máquinas, no depende de ningún software especial y es C, que es un lenguaje con el que en mayor o menor medida estamos familiarizados.

La compatibilidad incluye maquinas Unix, Windows NT y Linux, de memoria compartida o de memoria distribuida, mediante uso de hilos o MPI, de hecho fue escogido como parte de la batería de programas en el benchmark SPEC MPI2007.

Capacidades

El *render* se realiza partiendo de primitivas definidas, entre las que encontramos cajas, esferas o triángulos, todas estas primitivas provienen de un fichero de configuración de la escena, Tachyon tiene su propia sintaxis de definición de escenas, pero también es compatible con los formatos AC3 y NFF.

El producto de salida es codificado en Tga, pero tambien es compatible con JPG, BMP, PSD, SGIRgb e imap.

Funciona en 3 modos, shader bajo, shader medio y shader completo.

- Shader bajo

Solo se computan los objetos visibles, pero no se aplica modificadores ni texturas, por lo que solo se distinguen siluetas. Tiene un coste muy bajo tanto computacionalmente como en tiempo.

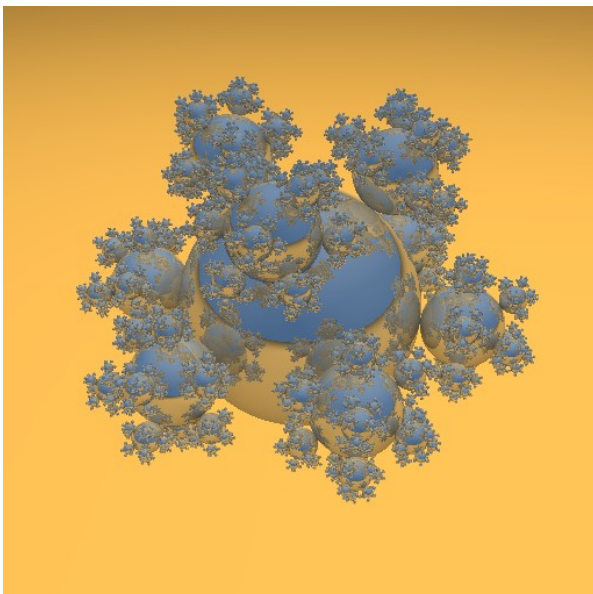


Fig 4.2 medium shader

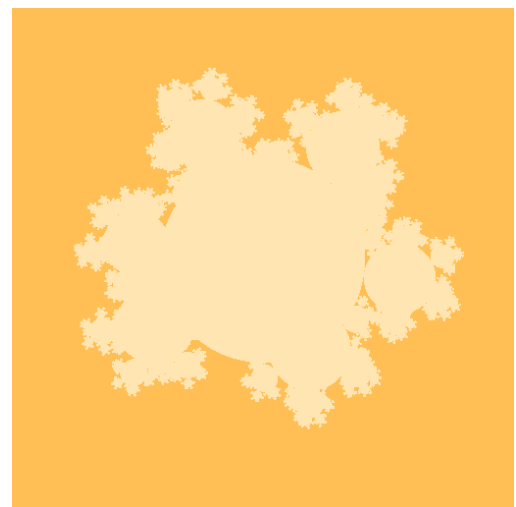


fig 4.1 low shader

- Shader medio

El shader medio computa por cada intersección el valor del color y lo acumula con los colores provenientes de las colisiones de los rayos reflejados y los transmitidos.

- Shader completo

El shader completo añade al anterior la capacidad de proyectar sombras y el brillos producidos por las luces.

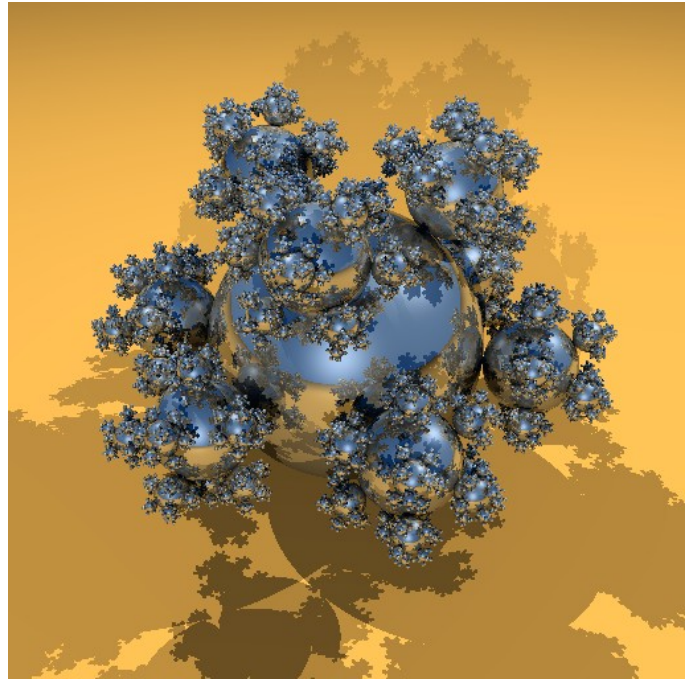


Fig 4.3full shader

Arquitectura

Tachyon está escrito con muchas opciones, para muchas máquinas y muy modular, por lo que puede resultar complicado de leer, pero tiene algunas soluciones muy elegantes para algunos problemas.

La idea es mantener todo muy modular y a pesar de que no se dispone de orientación a objetos en C, se realiza un trabajo parecido para definir los diferentes tipos de métodos. El computo aritmético para calcular la intersección entre un vector y un plano es diferentes al que se utilizaría para calcular el mismo con una esfera.

Sin embargo, el código no hace distinción y continua llamando a la función de intersección

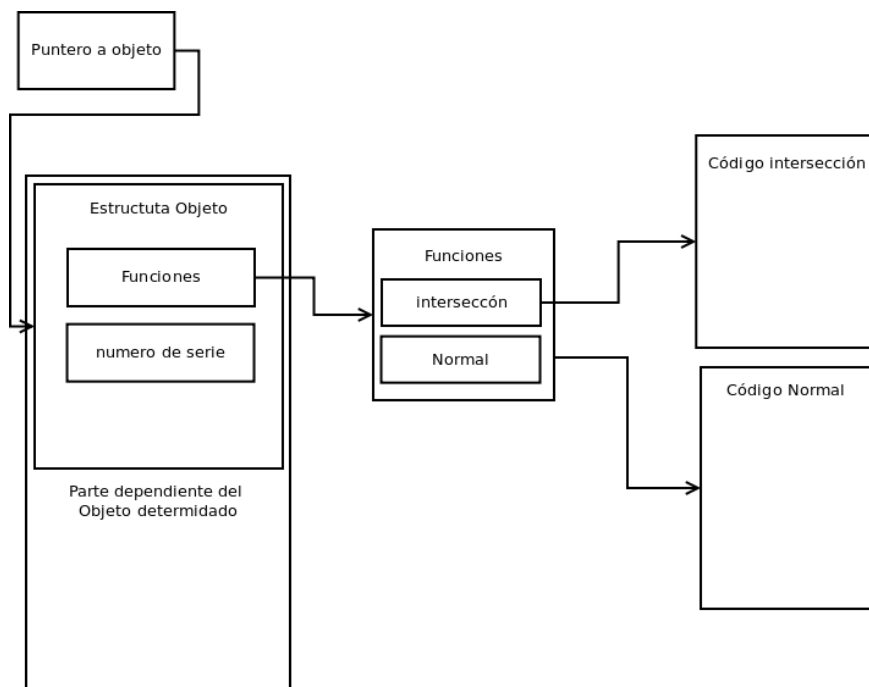


Fig 4.4 Arquitectura, objetos en Tachyon

sin parar a comprobar el tipo de objeto del que se trata, haciendo uso de un puntero a la función. Esta proporciona un gran nivel de abstracción, haciendo que olvidemos la matemática y podamos focalizar en el algoritmo en sí.

Para la definición del objeto, realiza un truco sutil pero muy eficiente. El objeto podría ser de diferentes tipos, y utilizar un puntero void para almacenarlos. Podríamos llamar a las funciones por su puntero asociado siempre y cuando el desplazamiento del puntero dentro de la estructura fuese siempre igual. Tachyon utiliza otra técnica, en la cual, se define un objeto general y una objeto para cada tipo, existe un macro, llamado `OBJECT_HEADER`, en el cual se define el área común a todos los objetos. De forma que se este macro se llama al definir cada objeto, por lo que el desplazamiento de cada uno de los elementos dentro de la estructura será el mismo, y podemos hacer un casting al tipo objeto sin problemas.

5 FASE 3: ADAPTACIÓN DE TACHYON A IRRADIA

Para realizar las modificaciones necesarias al comportamiento de Tachyon, fueron necesarios algunos cambios para asegurar la compatibilidad, reducir la complejidad del código y mejorar la gestión de los ficheros mediante un repositorio.

5.1 Makefile

Tachyon incorpora gran número de opciones para la compilación en diferentes arquitecturas y sistemas. Cada una de estas compatibilidades se garantiza con directivas de precompilador ubicadas en las llamadas al sistema susceptibles de ser modificadas.

La lista de compatibilidad es:

Versiones paralelas	Versiones Secuenciales
IBM AIX 5.x 64 bit, Pthreads ó MPI	IBM AIX 5.x
Intel ASCI Red MPI	OpenBSD/FreeBSD/NetBSD
IBM Blue Gene MPI	HP/UX 11.x
Cray J90,C90... Pthreads	SGI Irix 6.x
Cray T3E MPI	Linux
Cray XT3 MPI	Linux, AMD64/EM64T, GCC 3.x, 64-bit
Lemieux at PSC MPI	Linux, Intel C compilers, P4-optimized
HP/UX 11.x Pthreads	Linux using GCC 3.x Athlon optimizations
P-UX IA-64, HP C, Pthreads	Linux using GCC 3.x P4 optimizations
SGI IRIX 6.5.x Pthreads, 32 ó 64 bits	Linux, Portland Group Compilers, Pentium
Scyld Linux MPI	Linux, Portland Group Compilers, Athlon
Linux Alpha, Compaq C, MPI, QSWnet	Linux PowerPC
MPI (OSC LAM)	Linux Sony Playstation 2
Linux AMD64/EM64T, 64-bit, Pthreads ó MPI	Linux Alpha GCC o Compaq Compilers
Linux Pthreads Intel C compilers, o GCC	Linux IA-64, GCC
Linux IA-64, Pthreads, GCC o SGI compilers	Linux IA-64, SGI Pro64 Compilers
MacOS X PowerPC, Pthreads	MacOS X (aka Darwin, Rhapsody)
MacOS X Intel x86, Pthreads	Tru64 Unix, binary can run on AlphaLinux
Sun Solaris 9/10 ClusterTools 4.0 MPI	Sun Solaris 8/9/10
	Windows 95/NT with Cygnus EGCS/CygWin32

La lista es muy nutrida, pero nuestro objetivo inicial, Linux en IA-64 con paralelismo MPI no está, y como nuestra máquina no tiene los compiladores de SGI, debemos compilar con gcc.

Por ello se modificaron los ficheros Makefile, aún así resultaban engorrosos con tantas opciones y poco intuitivos para nuestros propósitos, por eso, finalmente se sustituyeron por un solo fichero Makefile

```

PROJECT=irradia
CC=cc
CFLAGS=-c -g -Wall -ffast-math
DEFINES=-DVMS -DLinux
LDFLAGS=-lm
EXECUTABLE=$(PROJECT)

SOURCES= src/api.c src/intersect.c src/texture.c src/apigeom.c src/light.c \
        src/threads.c src/apitrigeom.c src/parallel.c src/trace.c src/camera.c \
        src/rayqueue.c src/ui.c src/coordsys.c src/render.c src/util.c \
        src/global.c src/shade.c src/hash.c src/shade_iterativo.c \
        src/forms/box.c src/forms/plane.c src/forms/triangle.c \
        src/forms/cylinder.c src/forms/quadric.c src/forms/vol.c \
        src/forms/extvol.c src/forms/ring.c src/forms/grid.c \
        src/forms/sphere.c src/measures/measuring.c src/mathsvector.c \
        src/image/imageio.c src/image/jpeg.c src/image/ppm.c \
        src/image/sgirgb.c src/image/winbmp.c src/image/imap.c \
        src/image/pngfile.c src/image/psd.c src/image/tgafire.c \
        demosrc/ac3dparse.c demosrc/main.c demosrc/nffparse.c \
        demosrc/getargs.c demosrc/mgfpars.c demosrc/pars.c

OBJECTS=$(SOURCES:.c=.o)

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.c.o:
        $(CC) $(CFLAGS) $(DEFINES) $< -o $@

```

Para añadir compatibilidad MPI, se añade

```

DEFINES=-DVMS -DLinux -DMPI
LDFLAGS=-lm -lmpi

```

Para añadir compatibilidad con Pthreads, se añade

```
DEFINES=-DVMS -DLinux -DTHR
LDFLAGS=-lm -lpthread
```

5.2 Código fuente

El código fuente también fue modificado para simplificar todas las compatibilidades que no fueran necesarias, solamente se dejaron los fragmentos condicionados a la directiva MPI, para activar el soporte MPI y los correspondientes a Pthreads, ninguna otra tecnología de hilos se ha mantenido.

Entre las compatibilidades eliminadas, se encontraban todas las que hacían llamadas al sistema para arquitecturas específicas, como las llamadas para generar hilos de Microsoft o el paralelismo en máquinas Paragon. Así como optimizaciones que no se relacionaban con nuestro problema concreto.

Modificadores a la compilación

Se añadieron ciertos módulos que pueden ser compilados o excluidos según se desee:

- **NOMEDIDAS:** con la definición de este símbolo en el precompilador, se excluirá el código añadido para mostrar el reparto de carga y las estadísticas de trabajo.
- **RASC:** con la definición de este símbolo se activa la compatibilidad con la plataforma RASC y su dependencia de la librería librasclib.so

5.3 Eclipse

Para facilitar la programación, se migró todo el código a la plataforma eclipse, la cual hubo que configurar para poder compilar y enlazar el código.

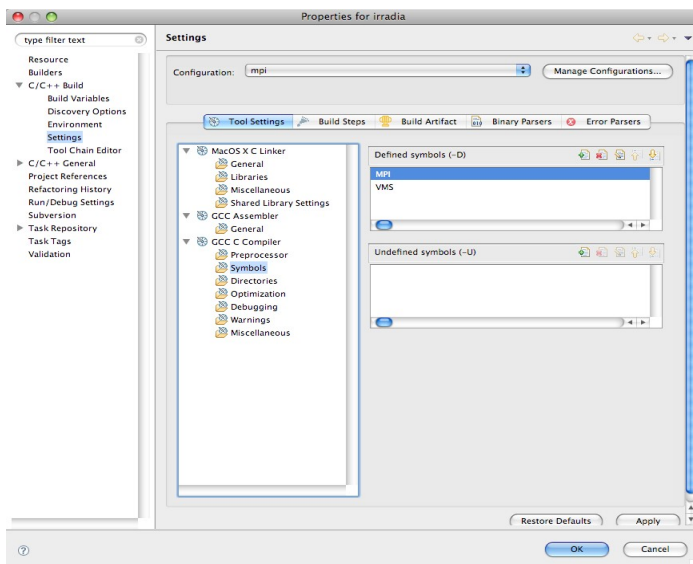


fig 5.1 configuración eclipse

Eclipse no es capaz de leer y entender el proyecto a partir de nuestro Makefile, por lo que hubo que trasladar el código y configurar este desde el principio.

En las propiedades del proyecto, las opciones que hubo que especificar fueron:

- **Symbols:** VMS y Linux para activar el código correcto.
- **Linker-Libraries:** m para enlazar la librería matemática.
- **Linker-Miscellaneous:** en Otros objetos, la ruta absoluta donde se encuentra nuestra librería dummyRasc.

5.4 Subversion

Para mejorar el trabajo en grupo sobre el código y la difusión de éste, se creó un repositorio utilizando el sistema subversión en un servidor gratuito de google.

<http://code.google.com/p/irradia>

En este se realizaron 23 actualizaciones significantes y se hizo uso de 2 ramas de desarrollo.

6. FASE 4: PARALELIZACIÓN EN IRRADIA

6.1 División del trabajo en Tachyon

Tachyon realiza el reparto de carga entre los procesadores o nodos de la siguiente manera:

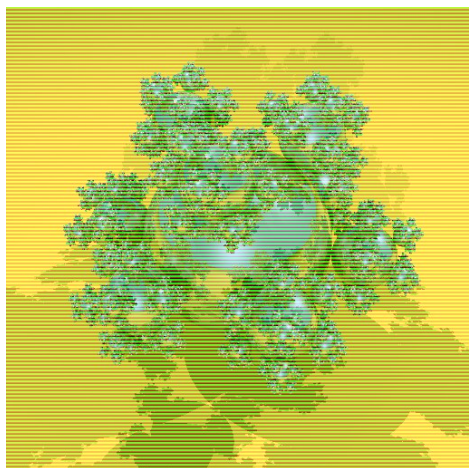


fig 6.1 division de trabajo

Se reparte por filas de un píxel de anchura, y se envían a procesar en los diferentes elementos de forma alternativa.

Si el reparto se hiciera por sectores, por ejemplo partiendo la figura en 4 áreas contiguas, podría darse el caso de que en un área apenas hubiera elementos mientras que otra estuviera fuertemente poblada, en este caso uno de los procesadores terminaría su trabajo y permanecería inactivo hasta que el otro terminara.

Como se puede observar en la imagen, para cuatro procesos, la imagen se divide por líneas entre los procesadores, lo cual hace un reparto del trabajo bastante justo en lo que a carga por procesador se refiere.

Si un píxel, en su primera intersección, encuentra un objeto lo más probable es que los píxeles adyacentes, encuentren el mismo objeto, por lo que el reparto es, en mayor o menor medida, equitativo, por lo menos para el primer rayo.

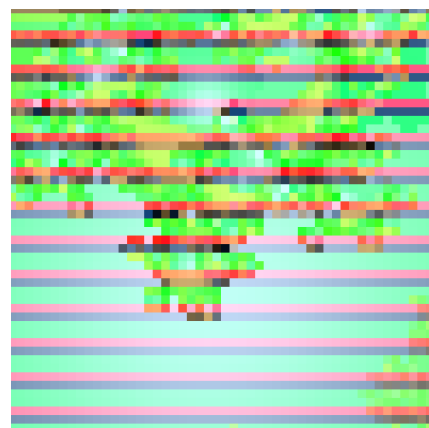


fig 6.2 detalle division de trabajo

Estas pruebas se realizaron añadiendo una opción al código, la cual se puede activar con un argumento al arrancar el código, `-diferenciate`, la cual activa modifica con una mascara de color las zonas procesadas por cada hilo.

6.2 Compartir los datos

El algoritmo es fácilmente paralelizable ya que existe un conjunto de datos de partida, la configuración de la escena, que no es modificada durante el proceso de renderizado, por lo que puede ser duplicado en nodos de trabajo (como en el modo de trabajo MPI) sin que sea necesaria ninguna comunicación para actualizarlos. O en el caso de la programación con hilos, se pueden leer simultáneamente sin el riesgo de estar leer datos sucios.

Existe un segundo conjunto de datos, que son locales al proceso en si, el rayo, los valores temporales, son datos que no salen una rama de ejecución en si, por lo que no existe ningún problema con estos tampoco.

Los datos de salida requieren un tratamiento especial, ya sea con MPI o con hilos, ya que todos ellos están destinados a escribir sobre el mismo área de memoria. En el caso de los hilos, la solución se realiza particionando el buffer de salida, de forma que un hilo solo escribirá dentro de

sus zonas de trabajo asignadas.

Con MPI, cada iteración que termina con una línea de píxeles, esta es enviada al nodo central. Las librerías MPI realizan la transferencia, pero el Proceso principal no recibe las líneas hasta que ha terminado su ejecución, donde se recogen y unen todas las líneas en un buffer, antes de codificarlo y escribirlo en disco. Por lo que toda esa información temporal permanece encolada dentro de la librería MPI, lo cual da tiempo de sobra a realizar las transmisiones, ya que no se requiere una interacción entre los procesos, y las latencias del medio serán poco importantes.

Propuestas de optimización

La solución de Tachyon es ingeniosa pero tiene un defecto, con este reparto conseguimos una carga uniforme garantizada para el primer rayo trazado por cada píxel en pantalla. Pero no podemos garantizar que permanezca equitativa después de los reflejos. Esto depende de la densidad de objetos en la escena así como de las posiciones de estos en esta.

Una solución para el reparto de trabajo sería almacenar los rayos en una cola, de forma que a medida que estos rebotan se encolan y los procesadores extraen rayos o bloques de estos de la cola de forma que todos tienen trabajo independientemente del origen del rayo.

El problema de esta solución es la coordinación entre los procesadores o incluso nodos en un sistema distribuido, por lo que podría generarse un problema mayor y no mejorar el rendimiento en absoluto. Esta es la razón por la que se respeta el algoritmo de Tachyon, por su sencillez y efectividad.

6.3 Versión recursiva en Tachyon y su equivalente Iterativo en Irradia

Esta modificación se realizó para familiarizarnos con el código de la aplicación y para incluir una cola que incorporara los rayos, para poder almacenarlos y procesarlos independientemente del dispositivo, ya sea un procesador o una FPGA.

Los rayos son encolados a medida que se avanza en el código, y acumulan el color de la intersección anterior.

Respecto al comportamiento general del programa, se consiguió una pequeña pérdida de rendimiento, después de modificar la cola y optimizar su lógica, reduciendo los saltos en el código para hacerlo lo más lineal posible, seguía siendo ligeramente más lento que el código original. Dicha diferencia se achaca a los saltos en el código que se pueden producir con las llamadas a la cola y a que el grado de recursión no es muy grande, solo hay, como máximo 6 llamadas recursivas (un árbol binario de profundidad 6 en caso de imágenes con transparencias).

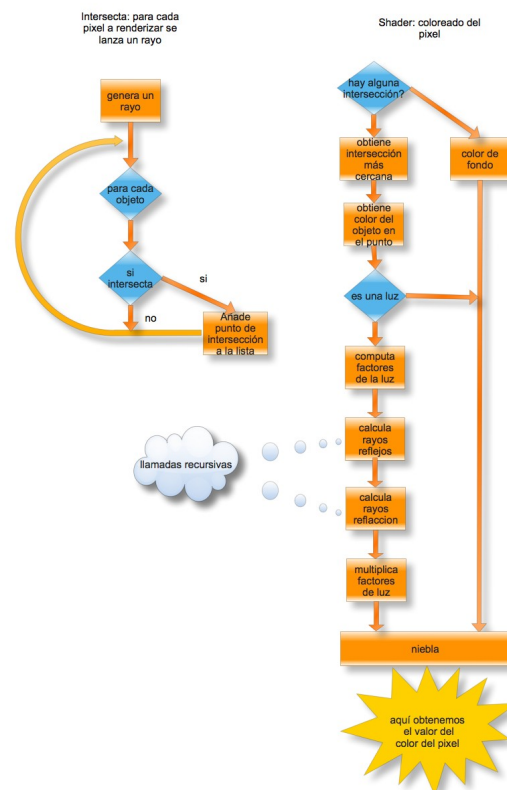


fig 6.3 ejecución recursiva

7 FASE 5: IRRADIA Y LAS FPGA's

Una vez estudiado el código, se procede a implementar parte del algoritmo en hardware, para de esta forma transferir carga de los procesadores a los dispositivos reconfigurables.

7.1 Diseño y algoritmo paralelizado

Para dividir la carga de trabajo y hacer uso de las FPGA's existían varias opciones que dependían de nuestra soltura con el VHDL y la máquina en general. En principio, cualquier algoritmo puede ser convertido en mayor o menor medida a código HDL, pero optamos por realizar un trabajo relativamente sencillo.

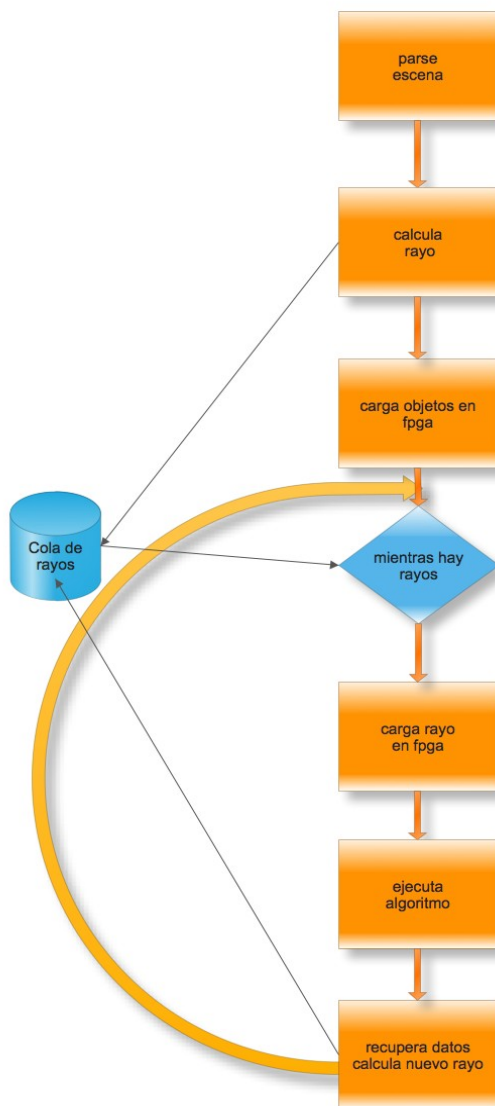


Fig 6.4 ejecucion iterativa

Los objetos se encuentran almacenados en una lista enlazada, de forma que cada vez que se

necesita comprobar una intersección, se recorre esta y se ejecuta la rutina adecuada dependiendo del tipo de objeto, los objetos que se pueden encontrar en la escena, implementados por Tachyon son:

CAJA
CILINDRO
VOLUMEN
REJILLA
PLANO
QUADRATICO
ANILLO
ESFERA
TRIANGULO

Lo optimo sería implementar todos ellos en el código HDL, pero los siguientes motivos, se optó por implementar solamente uno, el triángulo:

- Toda escena se puede descomponer en triángulos, se puede hacer una malla que cubra cualquier superficie fácilmente (existe software que lo hacen y es una opción muy común)
- La intersección entre un rayo y triángulo es relativamente sencilla. Solamente implica operaciones de suma, resta y multiplicación. Que son las más sencillas de implementar.

De esta forma se modifica el código de irradiar para almacenar los triángulos en otra lista, una lista que será cargada en la memoria de la FPGA al principio de la fase de renderizado y no será modificada en toda la ejecución, por lo que cada vez que sea necesario comprobar la intersección de un rayo con los triángulos, se actualizarán los registros para el rayo, y se lanzará el algoritmo en la FPGA mientras que el procesador comprueba el resto de objetos de haberlos.

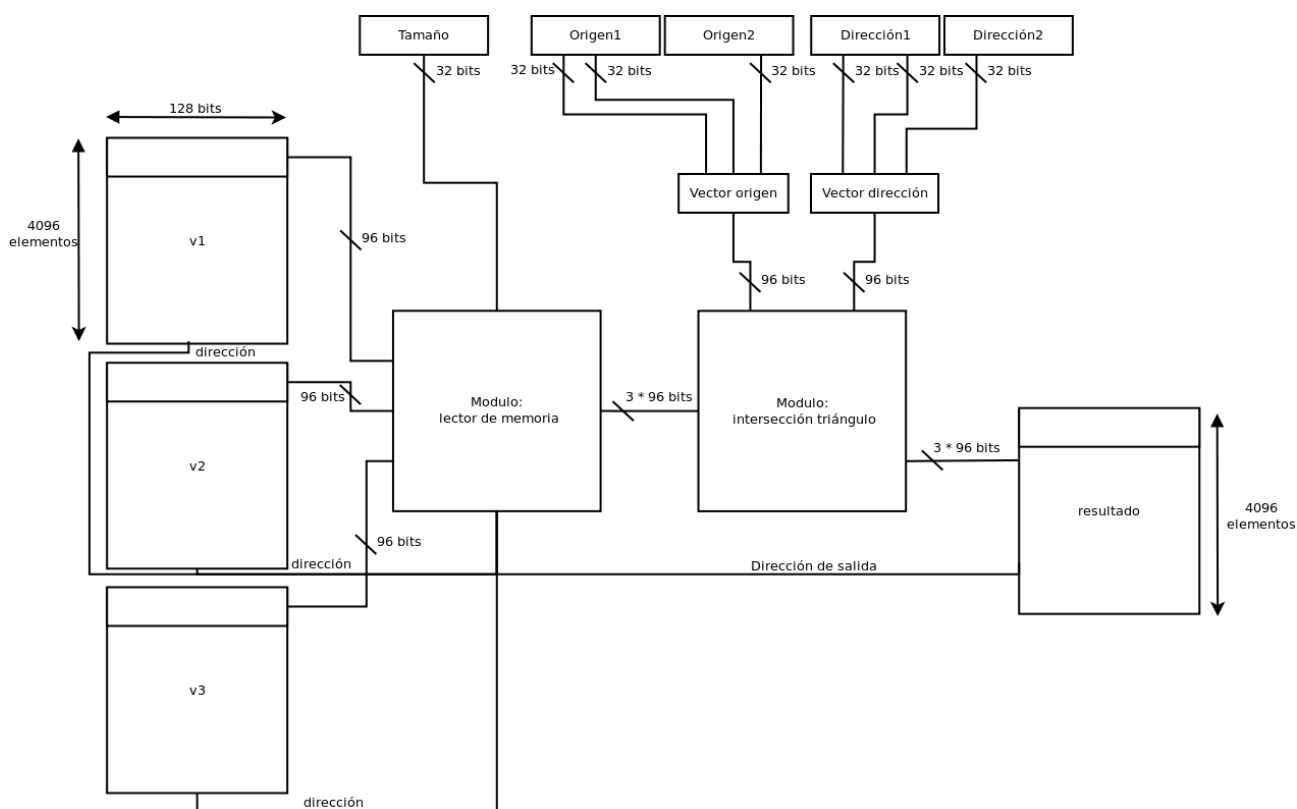
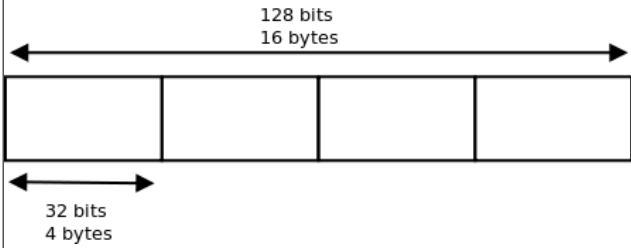


Fig 6.5 Datapath en nuestro algoritmo

Para poder transmitir los datos desde el código C a las FPGAs, es necesario formatear la memoria de estas a nuestro antojo, manteniendo un modo de acceso simple y eficaz. Como el bus de lectura de memoria es de 128 bits, nos permite leer un vector de 3 palabras de 4 bytes entero por ciclo, pero también sobra espacio (4 bytes) dentro de este, que será obviado. Por lo que se acuerda

un formateo de la memoria de la siguiente manera:

Código C	HDL
<pre> Typedef struct{ int x; int y; int z; int vacio; /* ignorar */ } vector128; </pre>	 <p>El diagrama ilustra la estructura de memoria en HDL. Muestra una fila horizontal dividida en cuatro secciones rectangulares iguales. Una flecha horizontal superior abarca toda la fila y está etiquetada como '128 bits' y '16 bytes'. Una flecha horizontal inferior abarca la primera sección y está etiquetada como '32 bits' y '4 bytes'.</p>

Para transmitir los vectores del rayo (dos, origen y dirección), hay que formatear los datos para que quepan en registros de 64 bits³. Por ello se utilizaron 2 registros por vector, dejando los 32 bits menos significativos.

El problema

Durante la implementación y pruebas de este algoritmo se observó una notable disminución del rendimiento que venía provocada por una optimización que no se había tenido en cuenta. Los objetos se preprocesan antes de comenzar el renderizado y se catalogan en áreas, de forma que los rayos comprueban primero si atraviesan dicha área antes de procesar los objetos en el interior, evitando de esta manera multitud de cálculos y acelerando el proceso global.

Esta optimización es completamente ignorada por todos aquellos objetos cargados en la lista alternativa, aquellos que se procesan en la FPGA, durante las pruebas sin RASC, se observó un rendimiento muy deficiente en el cálculo de triángulos, mientras que las esferas mantenían un comportamiento correcto.

³ Existe la posibilidad de modificar los datos en registros consecutivos, utilizando los llamados registros gordos (fat registers) pero no se utilizó por no estar correctamente documentados.

8. FASE 6: PROGRAMACION DE FPGAS

Simultáneamente con el análisis del software, se comenzó a desarrollar el código HDL para implementar el algoritmo en hardware.

8.1 Introducción a VHDL

Para programar dispositivos hardware como las FPGA's, necesitamos un lenguaje que nos permita describir la lógica que necesitamos abstrayéndonos de la circuitería electrónica. Esto es posible gracias a los lenguajes de descripción de hardware, HDL, del inglés *Hardware Description Language*. Además, estos lenguajes permiten jerarquizar y modularizar el código independientemente de la arquitectura sobre la que más tarde se implementará.

Es importante señalar, que el código VHD es completamente concurrente a menos que existan dependencias o tengamos estructuras de control, en cuyo caso, éstas irán siempre dentro de una estructura *process*, cuyo contenido se ejecuta secuencialmente. Aun así, las FPGA's pueden ejecutar más de una operación por ciclo de reloj gracias a las conexiones óptimas que se intentan realizar en el proceso de *Place and Route*, donde los elementos lógicos necesarios se mapean a los recursos físicos del dispositivo.

A la hora de realizar la implementación hemos tenido especial cuidado en evitar operaciones del tipo a (operador) b (operador) c... convirtiéndolas todas en binarias usando señales auxiliares para ello. Las operaciones de tipo no binario generan unos retardos. También se evita reutilizar las señales auxiliares para evitar complicar el cableado. Aunque todo esto aumenta el código, la propia herramienta de sintetización se encarga de optimizar todo lo posible el diseño. Todo el desarrollo de código se ha hecho con Xilinx ISE v11.

8.2 Algoritmo a implementar

Como se explicó en el capítulo anterior, se decidió implementar sobre hardware la intersección de rayos y objetos intentando conseguir que la FPGA realizara rápidamente operaciones algebraicas liberando de esa carga a los procesadores de propósito general del Altix.

Tras descartar la intersección de esferas por motivos que detallaremos en el apartado de problemas, decidimos implementar en VHDL la intersección de planos. El algoritmo de intersección del plano necesita como entradas el parámetro d , las componentes de la normal al plano $pln \rightarrow norm$ y los vectores de origen $ry \rightarrow o$ y destino del rayo $ry \rightarrow d$. A continuación se muestra el algoritmo en código C comentado. El código VHDL se encuentra en el CD adjunto⁴.

⁴ El código se puede encontrar en la ruta del CD software/irradia

```

/*
 * Cálculo de parámetros t y td.
 */
t = -(pln->d + (pln->norm.x * ry->o.x +
               pln->norm.y * ry->o.y +
               pln->norm.z * ry->o.z));

td = pln->norm.x * ry->d.x + pln->norm.y * ry->d.y + pln->norm.z * ry->d.z;

if (td != 0.0) //Si es 0, son paralelos y no existe intersección.
{
    t /= td;
    /*
     * Esta parte resultó problemática por la imposibilidad de dividir
     * dentro de la FPGA. Se optó por realizar esa división fuera y sustituir
     * la condición del IF por que ((td!=0) and (t > 0))
     */
    if (t > 0.0)
        ry->add_intersection(t, (object *) pln, ry);
    /*
     * La llamada a esta función se sigue manteniendo en el código
     * C. En la traducción a VHDL se activa una señal indicando
     * que el algoritmo ha finalizado con éxito
     */
}

```

Una vez implementado el plano, se vio que sería conveniente implementar la intersección con triángulos, pues una buena parte de las imágenes candidatas a tratar se encuentran generadas mediante multitud de ellos. El cálculo de la intersección con un triángulo es un caso particular de la intersección con un plano. La única diferencia es que el área de intersección es finita y con una forma definida, y no infinita como en el plano. El algoritmo que se aplica para este objeto consiste básicamente en estudiar primero si se produce una intersección con el plano que lo contiene, y a continuación determinar si el punto de intersección encontrado está dentro del área del triángulo. La intersección con este tipo de figuras contiene multitud de sumas y multiplicaciones pero nada más complicado. También existe una inversión que se solventó realizando el complemento a 2.

A continuación mostramos el algoritmo en lenguaje C comentado. Se aprecia que es necesaria la estructura del triángulo, con sus componentes *edge1*, *edge2* y *v0*, más los vectores origen y destino del rayo, *ry->o* y *ry->d*.

```

#define CROSS(dest,v1,v2) \
    dest.x=v1.y*v2.z-v1.z*v2.y; \
    dest.y=v1.z*v2.x-v1.x*v2.z; \
    dest.z=v1.x*v2.y-v1.y*v2.x;

```

```

#define DOT(v1,v2) (v1.x*v2.x+v1.y*v2.y+v1.z*v2.z)

#define SUB(dest,v1,v2) \
    dest.x=v1.x-v2.x; \
    dest.y=v1.y-v2.y; \
    dest.z=v1.z-v2.z;

#define EPSILON 0.00000005 /*Constante para omitir rayos que se acerquen a
la cara del triángulo*/

/*
 * Dentro del código se hace referencia a funciones que fueron sustituidas
 * por el autor por macros que aceleraban su código.
 * Son las referenciadas anteriormente y marcadas como #define.
 */

/*-----ALGORITMO DE INTERSECCIÓN-----*/

/* Cálculo del determinante */
CROSS(pvec, ry->d, trn->edge2);

/* Si el determinante es cercano a 0, el rayo está cerca del plano */
det = DOT(trn->edge1, pvec);

#if 0 /* define TEST_CULL if culling is desired */
    //Omitida la parte de TEST_CULLING
#else /* the non-culling branch */
    if (det > -EPSILON && det < EPSILON)
        return;

    inv_det = 1.0 / det;

    /* Cálculo de distancia de vert0 al origen del rayo */
    SUB(tvec, ry->o, trn->v0);

    /* Cálculo del parámetro U y comprobación de límites */
    u = DOT(tvec, pvec) * inv_det;
    if (u < 0.0 || u > 1.0)
        return;

```

```

/* Preparar parámetro V */
CROSS(qvec, tvec, trn->edge1);

/* Cálculo del parámetro V y comprobación de límites */
v = DOT(ry->d, qvec) * inv_det;
if (v < 0.0 || u + v > 1.0)
    return;
/*
 * Si los parámetros se encuentran dentro de la región,
 * el punto se haya entre los tres vértices.
 */

/* Cálculo del parámetro t, el rayo intersecta el triángulo */
t = DOT(trn->edge2, qvec) * inv_det;
#endif

ry->add_intersection(t, (object *) trn, ry);
/*
 * La llamada a esta función se sigue manteniendo en el código
 * C. En la traducción a VHDL se activa una señal indicando
 * que el algoritmo ha finalizado con éxito
 * /

```

El código generado se encuentra en el CD adjunto a la memoria⁵.

Con respecto al código en VHDL, todos los datos de entrada y el parámetro t de salida necesario para añadir el rayo a la cola de pendientes de procesar, los hemos representado con *STD_LOGIC_VECTOR (31 down to 0)*, es decir vectores de 32 bits. Esta elección se debe a que en el código original los datos venían en *float* ocupando 4 bytes. Además se tiene una señal binaria de salida que indica si hay o no intersección con los datos indicados.

Problemas

El cómputo en FPGA's tiene algunas limitaciones. La más notoria es la imposibilidad de dividir por números que no sean múltiplos 2. En caso de que sí lo sean, se puede implementar con desplazamientos. También es altamente costoso realizar la raíz cuadrada y fue éste el principal motivo por el que descartamos realizar la intersección de esferas.

Otro inconveniente que hemos tenido ha sido que los datos del programa original eran de tipo *float* y estaban comprendidos entre 0 y 1. Además teníamos una constante EPSILON con valor 5×10^{-8} . Al tener una aritmética lineal en el algoritmo de intersección, decidimos multiplicar por 10^8 los datos antes de entregarlos a la FPGA. El valor que devolvemos hemos de dividirlo por 10^8 de nuevo para tener una cifra coherente con el resto de datos del programa. Trabajar con enteros en formato *STD_LOGIC_VECTOR* resultó más sencillo y óptimo.

⁵ El código se puede encontrar en la ruta de l CD software/irradia/src/forms/triangle.c

9. FASE 7: GENERACION BINARIOS PARA ALTIX

La generación de los binarios (.bit) del algoritmo en vhdl o en verilog no puede ser sintetizada e implementada únicamente por las herramientas proporcionadas por Xilinx. Para que la compenetración con las librerías RASC y las FPGA's de la máquina Altix sea posible es necesario utilizar el paquete de desarrollo de FPGA's proporcionado por SGI: **RASC Algorithm FPGA Developer's Bundle** que contiene la herramienta: **RASC Algorithm Configuration Tool** (alg_conf_tool).

9.1 Entorno

La herramienta genera los ficheros necesarios para la sintetización e implementación de nuestro algoritmo incluyendo un makefile que realiza las llamadas necesarias. A la hora de generar los binarios para las FPGA's, es necesario tener en cuenta una serie de requisitos. Sin estos requisitos tanto la herramienta alg_conf_tool como el makefile no serán capaces de generar los binarios. Resumen de requisitos:

- Distribución Unix de 32 bits. (Debian, Red Hat, Fedora, Ubuntu...)
- Entorno gráfico o de escritorio (Gnome, KDE, XFCE,...).
- Librerías de programación Python.
- Xilinx ISE FOUNDATION 9.2i incluyendo software de síntesis XST.
- Configuración de variables de entorno: Rasc y Xilinx.

El primer requisito es tener una versión Linux Unix de **32bits**, puesto que XILINX no funciona en versiones de 64 bits, nosotros hemos utilizado la versión 9.10 - Karmic Koala - publicada en Octubre de 2009 y soportada hasta Abril de 2011. Aunque también hemos implementado los archivos con la versión 10.04 sin problemas. También es necesario un **Shell**, cualquier Unix lo trae consigo, utilizamos la versión 1.0 que trae la distribución de Ubuntu mencionada anteriormente.

La versión 2.2 de las librerías Rasc fueron implementadas para versiones 9.2 del Xilinx ISE. Lamentablemente no soporta versiones superiores a pesar de que en la guía de usuario de SGI lo mencione. Hay que tener en cuenta que la versión gratuita de XILINX, la versión Webpack 9.2i, no soporta la arquitectura de las FPGAS VIRTEX4. En concreto las de nuestra máquina Altix: **VIRTEX4 XC4VLX200** solo son soportadas en versiones ISE Foundation, las cuales no era posible descargar durante el inicio del proyecto. Xilinx empezó a publicar versiones **ISE Foundation** (completas) anteriores a las actuales desde el inicio del año 2010. Fue en ese momento en el que pudimos adquirir la versión 9.2 con licencia de estudiante de pruebas durante un mes. La máquina Altix no provee de ninguna licencia ni versión de Xilinx a pesar de necesitarla, como se explica a continuación, para la generación de los archivos binarios y de configuración de las FPGAS.

Otro factor importante es que es necesario un **entorno gráfico** o de escritorio para el uso de la herramienta de configuración de algoritmos (alg_conf_tool). Nosotros utilizamos la versión del Gnome instalada en el Ubuntu, en concreto la versión 2.28.1. Para la generación del archivo de configuración es necesaria tener instalada una versión de las librerías **Python**. Nosotros usamos la versión Python 2.6.4.

Para la síntesis es posible utilizar otro sintetizador a parte del que trae Xilinx ISE (**XST**) denominado Synplicity Synplify el cual no es posible conseguir licencia de estudiante ya que solo existe en versión de pago.

nos ayudaron durante la generación de los archivos binarios correctos para el proyecto.

Las librerías de Python son utilizadas para recorrer los archivos .vhd y .v en busca de los extractors (definiciones entrada/salida del usuario) y generar los archivos de configuración.

Posteriormente teóricamente es posible la depuración mediante registros de salida de el algoritmo unidos a un comando añadido al gdb de Unix, aunque nuestra máquina no lo llevaba instalado durante la realización del proyecto y no nos lo permitía.

9.3 RASC Algorithm FPGA Developer's Bundle

El paquete lo proporciona SGI, y está contenido en el CD. Tan solo es necesario la descompresión del archivo *ia32_rc100_22_dev_env.tar.gz* para empezar a utilizarlo. El entorno está dividido en varias carpetas y subcarpetas. El siguiente esquema simplificado representa los archivos útiles para la generación de binarios:

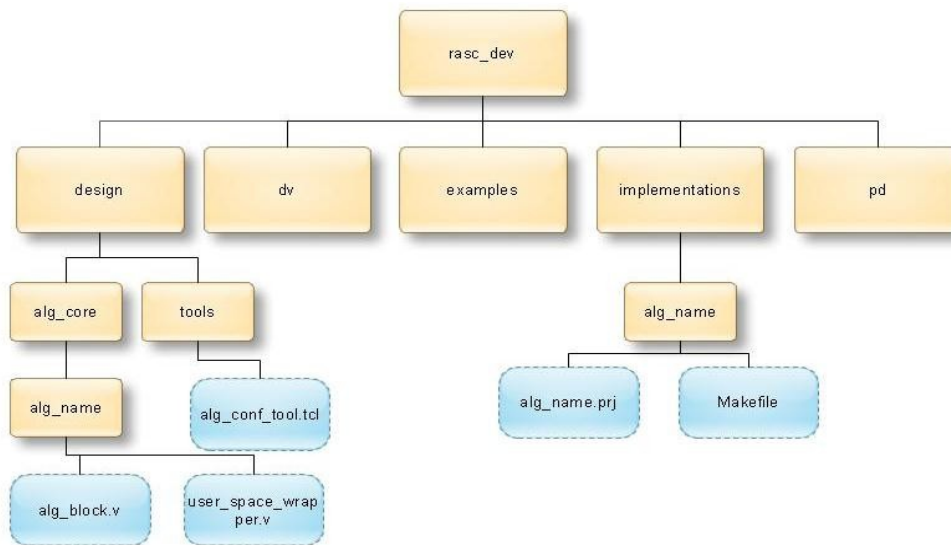


fig 9.2 developers bundle

La carpeta Design: contiene la carpeta `alg_core` que es donde se crea el archivo `<alg_block.v>`, entre otros, para cada algoritmo que generemos con la herramienta `alg_conf_tool`. En el apartado estructura de `<alg_block.v>` se explica la utilidad del fichero. Además contiene la carpeta `tools` que es donde se ubica la herramienta `alg_conf_tool`.

La carpeta Implementations contiene los archivos para cada algoritmo generado por `alg_conf_tool`. Los que nos interesan son el `<algoritmo.prj>`, `<Makefile>` y `<Makefile.local>`. En esta ubicación es donde nosotros nuestro diseño vhd.

La carpeta de examples: contiene ejemplos útiles desarrollados en vhd, verilog y preparados para las herramientas de síntesis xst y simplify.

RASC Algorithm Configuration Tool

La herramienta proporcionada por SGI `rasc_alg_conf_tool` es en realidad un asistente que permite generar las carpetas y archivos en el paquete del desarrollador de FPGAS con RASC correspondientes con nuestros algoritmos. El asistente se ejecuta con el script ubicado en la ruta `rasc_dev/design/tools`.

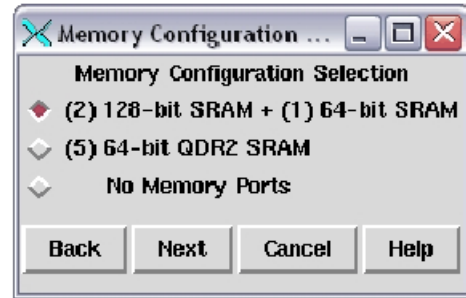


fig 9.3 y 9.4 asistente para la configuración

Realizando los pasos del asistente nos generará la carpeta con el nombre de nuestro algoritmo en `rasc_dev/implementations` y en `rasc/dev/design/alg_core`. En la carpeta `implementations/alg_name` es donde ubicaremos nuestro diseño en vhd.

Para generar los archivos binarios es necesario añadir en el archivo `alg_name.prj` la línea `vhdl work ./alg_name.vhd` para indicar así al XST que compile nuestro algoritmo. Una vez concluido el asistente, modificado el archivo `<alg_name.prj>` y copiado en la misma carpeta nuestro algoritmo `<alg_name.vhd>` solo se requiere la ejecución del `<Makefile>` por línea de comandos para empezar la generación de los archivos. En el apartado Ejecutando el Make se narra el transcurso de llamadas que el script realiza.

Flujo de la implementación según la herramienta

El siguiente esquema refleja el flujo de implementación de binarios (.bin) y de los archivos de configuración (.cfg). Para que los archivos de configuración sean correctamente creados ha de llamarse al Makefile que genera la herramienta/asistente. El Makefile contiene las rutas de los archivos y genera distintas llamadas a Xilinx ISE, XST y Python.

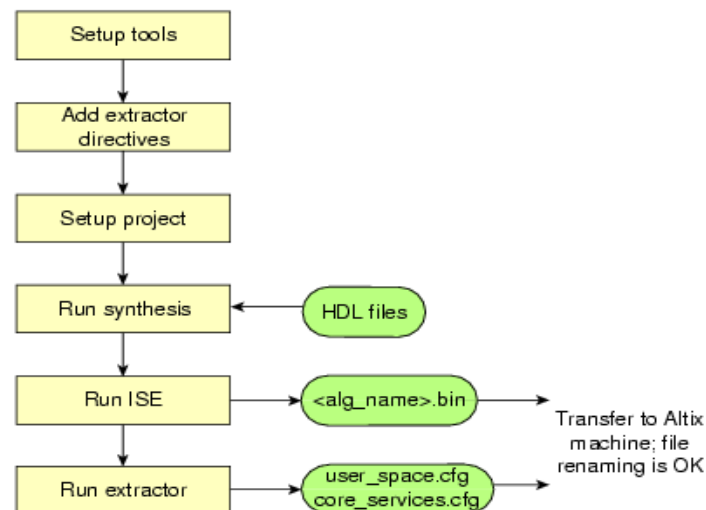


fig 9.5 flujo implementación

9.4 Compilación y síntesis

Para la generación de nuestro algoritmo (intersección de triángulos) tuvimos que indicar la velocidad de reloj mínima posible, en concreto una velocidad de 50 Mhz, con un valor de reloj más alto (100,150 o 200Mhz) durante la implementación externa del algoritmo Xilinx ISE nos avisaba que no sería capaz de generar todas las operaciones en un ciclo de reloj que era nuestro propósito.

En el último formulario del asistente permite añadir un reloj suplementario, pero nuestro diseño no lo requería.

Desde software accedemos a memoria una sola vez, en el inicio de la llamada para guardar los valores de los vértices del triángulo, por lo que no nos era necesario contar con buffers streaming de acceso directo a memoria. Solo necesitamos 2 memorias una de entrada y otra de salida (128 bits), por lo que seleccionamos esa opción en el asistente. Existe otra opción de 5 memorias de 64 bits.

Lo que sí modificamos son los registros que contienen los rayos de origen y de destino por eso decidimos que necesitaríamos un total de 6 registros entrada/salida para almacenar cada una de las componentes (x,y,z).

Era inútil tener registros de depuración o debug pues ya sabíamos que nuestro servidor Altix no tenía instalado las herramientas del gdb para RASC.

Estructura de alg_block.v

Alg_block.v contiene el módulo en verilog que conecta nuestro diseño en vhd1 con las FPGAS. Se genera automáticamente aunque es necesario añadir un módulo que conecte con tu diseño vhd. En el módulo es posible renombrar variables.

El fichero se puede dividir en varios campos:

- Módulo top-level que conecta directamente con la FPGA.
- Definiciones de los puertos del módulo.
- Asignaciones por defecto de puertos.
- Módulo propio.

En las definiciones de los puertos es posible asignar un valor predeterminado al paso de reloj (de subida o de bajada), valor por defecto de los registros y de los demás puertos. En nuestro diseño queríamos direccionar nuestra memoria por bytes por lo que colocamos a la direcciones de memoria 4 bits por la derecha para así no tener que acceder a esas direcciones.

Extractors de nuestro diseño

Los extractors son porciones de código comentadas que indican a RASC que puertos de entrada/salida utilizaremos desde software para acceder al diseño VHD. Éstos se ubican en los ficheros vhd1 o verilog de nuestro algoritmo. También se puede indicar la versión del algoritmo vhd1 y del módulo superior del diseño.

Para nuestro diseño utilizamos los siguientes extractors:

```
//-- extractor VERSION: 0.4
//-- extractor CS: 2.1
//-- extractor SRAM:a_in 8192 128 sram[0] 0x00000 in u stream
//-- extractor SRAM:b_in 8192 128 sram[0] 0x10000 in u stream
//-- extractor SRAM:c_in 8192 128 sram[0] 0x40000 in u stream
//-- extractor SRAM:d_out 8192 128 sram[1] 0x00000 out u stream
//-- extractor REG_IN:r_orx 64 u alg_def_reg[0][63:0]
//-- extractor REG_IN:r_ory 64 u alg_def_reg[1][63:0]
```

```
//-- extractor REG_IN:r_orz 64 u alg_def_reg[2][63:0]
//-- extractor REG_IN:r_dex 64 u alg_def_reg[3][63:0]
//-- extractor REG_IN:r_dey 64 u alg_def_reg[4][63:0]
//-- extractor REG_IN:r_dez 64 u alg_def_reg[5][63:0]
```

Extractor Version: para indicar la versión del algoritmo.

Extractor CS: para indicar la versión de los core services.

Extractor SRAM: vectores de entrada y salida del algoritmo que se ubican en la memoria ram correspondiente [i]. con 8192 elementos de 128 bits cada uno. Se debe indicar la primera dirección de memoria donde empieza el vector. Si es de entrada (in) o de salida (out) y si es un valor fijo (fixed) o que cambiaremos durante la ejecución de las llamadas rasc desde C (stream).

Extractor REG_IN: registros de entrada de 64 bits donde almacenaremos cada uno de los componentes de los rayos .

Por tanto nuestro diseño contiene una memoria de entrada (128 bits): donde almacenamos los valores de los vértices de los triángulos de la escena (3 vértices de 3 componentes cada uno) almacenamos el primer vértice en A, el segundo en B y el tercero C. Puesto que en nuestro diseño necesitaremos los 3 vértices y los rayos de origen y destino para saber si interseccionan o no.

Ejecutando el MAKE

El makefile generado por el asistente viene preparado para compilar cualquier algoritmo para la máquina y las FPGAs desde consola en batch mode. Permite su configuración desde el fichero Makefile.local. Al inicio del fichero ya se encuentra la configuración de nuestras FPGAS como se observa a continuación:

```
# Define FPGA used
FAMILY=VIRTEX4
PART=XC4VLX200
PACKAGE=FF1513
SPEED=10
```

El Makefile realiza una serie de llamadas al Xilinx ISE, xst y otros comandos de unix:

```
make oneloop : realiza la implementación de Xilinx con las llamadas:
    make ngdbuild
    make map
    make par
    make ncd
    make trce
make bitgen : genera el .bit
make extractor : genera los archivos de configuración.
```

Una vez realizado el make (tarda alrededor de 40 minutos en un ordenador con procesador Intel Core 2 Duo a 2,6 Ghz y 4 GB de RAM) tendremos los archivos binarios y de configuración listos para transferir al servidor y ejecutarlos desde código C. los archivos de configuración de usuario y de los servicios del core se generan en la carpeta de implementations/alg_name. Y El fichero .bit se ubica en la carpeta /rev1 dentro de la anterior carpeta.

Transfiriendo los algoritmos al servidor Altix: devmgr

Una vez compilado nuestro algoritmo lo transferimos a la máquina Altix con el protocolo sftp que teníamos habilitado. Para mover los archivos binarios y de configuración a un registro desde el cual las FPGAs puedan ser configuradas es necesario utilizar un demonio que provee SGI denominado devmgr.

Los distintos comandos que hemos utilizado para este fin se describen a continuación.

devmgr {-a}	Añade algoritmo al registro
{-d}	Elimina el algoritmo del registro
{-u}	Actualiza el algoritmo con el nombre indicado
{-q}	Consulta los algoritmos que existen en el registro
{-i}	Enumera las FPGAs disponibles
{-l}	Carga un algoritmo en la fpga
{-v}	Visualiza version del comando
{-m avail unavail}	Marca la disponibilidad de una FPGA
{-m res rel}	Reserva o libera una FPGA
{-x on off}	Enciende el servidor de depuración
[-n nombre]	para indicar nombre del algoritmo
[-b ruta fichero]	para indicar nombre del binario (bittstream .bit)
[-c ruta fichero]	para indicar nomnre del archivo de configuracion (.cfg)
[-s path_name]	para indicar nombre de la configuracion del core services (.cfg)
[-y on off]	Enciende la depuración

Para añadir nuestro algoritmo utilizamos el comando:

```
devmgr -a -n triangulos -b triangulos.bit -c triangulos_config.cfg -s triangulos_services.cfg
```

Una vez subido el algoritmo al registro, ya podemos configurar las FPGAs con él y ejecutarlos desde código C con las librerías RASC.

9.5 Problemas y soluciones

A lo largo de la fase de la generación de los binarios del algoritmo sufrimos una serie de problemas derivados por varios factores. Uno de ellos y el más importante es la falta de documentación que existe sobre la máquina y en concreto en la generación de los .bit. La única referencia que encontramos era el manual oficial de SGI.

Los primeros problemas que tuvimos estuvieron relacionados con la compatibilidad de la herramienta alg_conf_tool y sus requisitos, por eso hemos creído conveniente empezar el capítulo

describiendo los requisitos necesarios para su generación. Más tarde, con la herramienta `alg_conf_tool` funcionando correctamente nos encontramos con errores que describimos a continuación y nuestras soluciones a ellos con el fin de servir como experiencia a futuras implementaciones con el servidor Altix.

Problemas al realizar Make

El Makefile que genera la herramienta de configuración puede editarse, aunque no es necesario siempre y cuando se haya elegido las opciones correctas en el asistente `alg_conf_tool`. También permite su configuración en el archivo `<Makefile.local>` pudiendo alterar rutas de ficheros o variables de entorno entre otros.

Es posible que durante la ejecución del Makefile aparezcan warnings derivados de la compilación y síntesis de los procesos al que hace referencia de Xilinx. Aunque estos no suponen una incorrecta generación de los archivos finales. Existe un momento en el que el script del archivo `<Makefile>` realiza un `cat` que puede que no reconozca donde se encuentran unos determinados ficheros a pesar de tener las variables de entorno y las rutas a éstos correctamente

Descripción del error:

```
user@user-laptop:~/rasc_dev/pd/constraints/ise$ cat virtual_gnd.ucf ssp.ucf
prohibit.ucf qdr_bank0.ucf qdr_bank1.ucf qdr_bank2.ucf qdr_bank3.ucf
qdr_bank4.ucf qdr_misc.ucf qdr_0_2x_use.ucf > prueba.ucf
cat: virtual_gnd.ucf: No existe el fichero ó directorio
cat: ssp.ucf: No existe el fichero ó directorio
cat: prohibit.ucf: No existe el fichero ó directorio
cat: qdr_bank0.ucf: No existe el fichero ó directorio
cat: qdr_bank1.ucf: No existe el fichero ó directorio
cat: qdr_bank2.ucf: No existe el fichero ó directorio
cat: qdr_bank3.ucf: No existe el fichero ó directorio
cat: qdr_bank4.ucf: No existe el fichero ó directorio
cat: qdr_misc.ucf: No existe el fichero ó directorio
cat: qdr_0_2x_use.ucf: No existe el fichero ó directorio
```

Solucion al problema:

copiamos `prueba.ucf` a la carpeta y hacemos el `cat` manual luego podemos continuar la ejecución del `<Make>` sin tener que realizar de nuevo la compilación inicial realizando la llamada correspondiente, es necesario el análisis del Make para no saltarse ningún comando.

eliminamos del makefile:

```
cat ${UCF_FILE_LIST} > ${SYNTHESIS_PROJ}.ucf
```

volvemos a ejecutar el make

Problemas con el Reloj

Descripción del error:

```
im INFO:Timing:3284 - This timing report was generated using estimated delay
```

information.

For accurate numbers, please refer to the post Place and Route timing report.

En nuestro diseño solo era posible la generación del hardware lógico con un reloj de 50 Mhz pues el algoritmo realiza una serie de operaciones que no era posible realizarla en un ciclo de reloj con más tiempo. Más tarde este problema derivó en otro que se explica en el capítulo Test.

Problemas con los extractors

A pesar de que en el manual de SGI RASC indica que es indiferente colocar los extractors en el diseño VHDL o en el módulo superior (alg_block.v) nuestro archivo de configuración no se realizaba correctamente colocándolo en el vhdl.

Tuvimos que colocarlo en el módulo de alto nivel para que los comandos de Python del makefile lo reconociesen.

10. FASE 8: SIMULACIÓN POR SOFTWARE

Durante el desarrollo del proyecto se vio la necesidad de probar el código C, y poder continuar la implementación del software sin verse lastrada esta parte por el lento avance de la parte del proyecto destinada al hardware. La solución vino de la mano de un subproducto, el cual no entraba en los planes originales del proyecto pero ha resultado útil y novedoso.

10.1 DummyRASC

DummyRASC⁶ es una librería que cumple con el objetivo de tener una librería compatible con la librería RASC de SGI, de forma que se pueden implementar las llamadas a ésta, y compilar en cualquier máquina, la librería está programada utilizando solamente librerías POSIX y estándar. Por lo que los desarrollos pueden hacerse en cualquier plataforma compatible con éstas⁷ donde no tenemos la librería original. Podremos ejecutar el código y conseguir un comportamiento similar, a nivel lógico.

De esta forma podemos probar la lógica y comportamiento del software antes de que el hardware esté listo. Una vez programado, tan solo con cambiar la librería dinámica que se enlaza con nuestro software por la librería RASC original, tendremos la funcionalidad completa, ya que se han conservado los prototipos y comportamientos tal y como dice el manual.

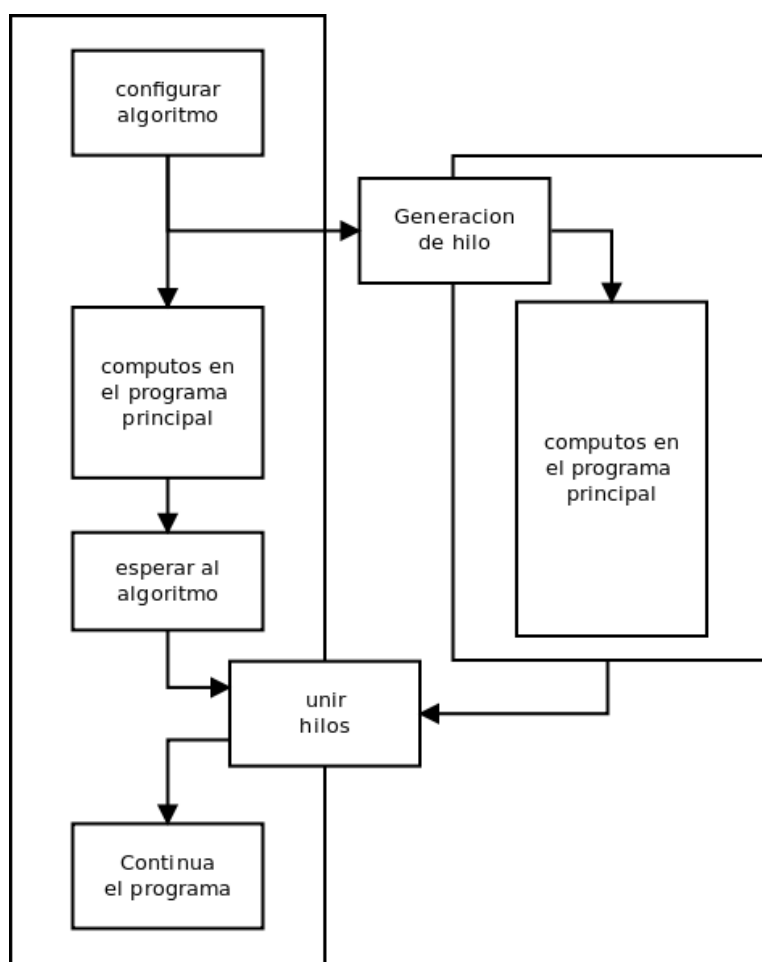


fig 10.1 uso de DummyRASC

De esta manera, se puede mantener a dos grupos de desarrollo, uno de software y otro de ingenieros hardware, trabajando en la aplicación de forma paralela y agilizando el proceso de desarrollo.

La idea principal es tener el código C que realiza la tarea que se va a traducir a hardware encapsulado en un dispositivo virtual. De forma que se interactúa con la librería como si se estuviera usando una FPGA real, pero en realidad se está expandiendo un hilo que ejecuta el algoritmo tal y como está escrito en C de forma concurrente. Obviamente no se obtiene ninguna mejora de rendimiento, solo se consigue simular el comportamiento de la librería RASC.

⁶ DummyRASC se proporciona en el CD, con un completo manual explicando su arquitectura, funcionamiento y cambios futuros necesarios.

⁷ Unix, probado en Linux (Ubuntu) y MacOS X

10.2 Utilización de la API

Las funciones que proporciona la librería son idénticas a las que proporciona la API original, y tienen un comportamiento similar, por lo que el manual de ayuda de RASC escrito por SGI es una buena referencia para programar DummyRASC.

La diferencia radica en que solo se implementaron ciertos métodos, que son los que se utilizan en los ejemplos y los que se usan en nuestro desarrollo.

int rasclib_resource_reserve (ncops, reserve_name)

Reserva un numero de FPGAs para nuestro algoritmo, en principio se dispone de un numero determinado de FPGAs disponibles, fijado dentro de la librería y modificable cambiando el código, en esta version se proporcionan 4, como en nuestra máquina.

int rasclib_resource_configure (al_id, ncops, reserve_name)

Configura un numero de FPGAs, de las previamente reservadas, y que no tiene por que ser el mismo, aunque, obviamente debe ser menor. Al configurar se leen las especificaciones del algoritmo escogido, que esta implementado internamente dentro de la librería, y se hacen las comprobaciones y reservas de memoria necesarias.

int rasclib_resource_return (al_id, ncops)

Devuelve las FPGAs configuradas a nuestro almacén de reservadas, para poder utilizarlas con otro algoritmo si se quiere.

int rasclib_resource_release (ncops, reserve_name)

Se liberan las FPGAs y vuelven a estar disponibles para todos los usuarios, al tratarse de FPGAs virtuales, no existe la necesidad de compartir un dispositivo hardware con ningún otro programa, por lo que no es una función critica, pero es importante incluirla cuidadosamente ya que si lo es en el sistema real.

int rasclib_algorithm_open (al_id, flags)

Una vez configurado el algoritmo, este se abre, como si de un fichero se tratase, para tener acceso a su areas de memoria y a los registros mapeados en esta.

int rasclib_algorithm_send (algo_desc, algo_array_id, buffer, count)

Cargamos un vector de datos en una de las areas descritas por el algoritmo. Estas areas son accedidas por nombre, que se le asigna dentro de la librería y en las directivas Extract del HDL en la librería RASC original.

int rasclib_algorithm_receive (algo_desc, algo_array_id, buffer, count)

Enlazamos un buffer a la salida de los datos, de forma que tendremos la salida en este, se comporta exactamente igual que los bufferes de entrada con respecto a los nombres. El algoritmo utilizará este buffer directamente en la versión DUMMY, por lo que todos los calculos se realizarán contra este area directamente en el hilo secundario.

int rasclib_algorithm_reg_write(al_desc, alg_reg, data, ndata_elements, flags)

Se envía un registro a alguno de los registros especificados en el algoritmo, todos los registros son de 64 bits, esto es 8 bytes o long long en C.

int rasclib_algorithm_get_num_cops(al_desc)

Devuelve en número de FPGAs involucradas en un algoritmo determinado.

int rasclib_algorithm_go (al_desc)

Lanza el algoritmo en el dispositivo, en realidad, la versión real no puede garantizar que esto ocurra hasta el commit, ya que todos los comandos se ejecutan encolados, y debemos esperar a que se satisfagan los anteriores primero, a medida que el bus esté disponible, por ello en la versión simulada no se lanza el hilo hasta el commit. Aún así es imprescindible llamar a go.

int rasclib_algorithm_wait (al_desc)

Espera hasta que el algoritmo haya terminado en la FPGA, puede ser que este ya hubiera terminado antes de llegar a wait, de esta se continua la ejecución sin espera ninguna.

int rasclib_algorithm_commit (al_desc, algo_callback)

Commit fuerza la finalización de la cola de comandos, la librería de simulación no encola los comandos porque no es necesario, todas las llamadas son síncronas, por lo que commit solamente se encarga de lanzar el hilo si se ha llamado a go previamente.

int rasclib_algorithm_close (al_desc)

Cierra el manejador de un algoritmo, y con ello devuelve todas las FPGAs involucradas al almacén de reservadas. Libera la memoria reservada.

void rasclib_perror (string, ecode)

Devuelve un mensaje de error dependiendo del código proporcionado.

11. FASE 9: INTERCONEXIÓN FPGAS

La parte más delicada de la programación C es la conexión con las FPGAs, su configuración y la carga de datos. Hay que tratar con mucho cuidado todos los datos por el alineamiento de la memoria, el orden de significancia de los bits y el tipo de los datos, además de hacer las llamadas a la api de forma ordenada y correcta.

Para ello se implementó un módulo nuevo, que funciona en forma de API. Y que intenta ser lo más modular posible y sin tener en cuenta una implementación determinada, para hacer los cambios lo más sencillos posibles.

11.1 Llamadas y módulos

Todo el modulo ha sido diseñado pensando en los tipos definidos en el programa y en las necesidades de este, manteniendo un número bajo de llamadas para no complicar tampoco el código allí donde se haga una llamada.

Los datos que se cargan en las FPGAs son copiados en búfferes intermedios que se mantienen sin liberar hasta que se realiza el commit, de forma que aseguramos que la dirección de memoria este intacta mientras los comandos estan encolados, ya que no podemos garantizar que estos se hallan copiado a las memorias de las FPGAs hasta que se llama a commit. Por ello se almacenan estos punteros para poder liberarlos en caso de que algo falle, se mantiene una estructura de control que mantiene las referencias, por ello es necesario llamar a la función de inicialización y a la de liberar recursos.

void rcnc_init ();

Inicializa el módulo, prepara los recursos y estructuras de control.

int rcnc_monta_entorno (int numFPGAS, int alg_code);

Prepara el entorno para trabajar con RASC, esto implica reservar los dispositivos, configurarlos y abrirlos. Dejando todo listo para la carga de datos y su utilización.

int rcnc_load_objs (int alg_code, object *objs, int num);

Carga los objetos de determinado tipo en la memoria del dispositivo, esta es la única funcion que tiene un comportamiento diferente dependiendo del objeto que se haya implementado en HDL, ya que cada uno tienen unas características diferentes y por ello la carga de datos es diferente. También se actualiza el número de objetos para reducir el número de iteraciones en la FPGA, este ultimo se pasa en forma de registro.

int rcnc_load_ray (int alg_code, ray *rayo);

La carga del rayo se hace a traves de 4 registros, dos para vector del rayo, que son el origen y la dirección.

int rcnc_go (int alg_code);

Se arranca la máquina para comenzar a ejecutar el algoritmo. Este método implica una llamada a go y una llamada a commit. También borra los búfferes intermedios una vez ya no son necesarios.

int rcnc_wait (int alg_code);

Espera hasta que el algoritmo termina, el hilo de ejecución en la CPU se bloquea hasta que la FPGA termine, por lo que se puede posponer mientras haya calculo que realizar en esta.

```
int rcnc_set_result_ptr (int alg_code, void *buffer, int num);
```

Enlazamos un puntero a un area de memoria deseada para recibir el resultado del computo en esta.

```
int rcnc_desmonta_entorno(int alg_code);
```

Libera el entorno, la memoria y los recursos.

```
void rcnc_close_all ();
```

Libera las estructuras de control y finaliza.

11.2 El tratamiento de la memoria en el caso de los triángulos

Los objetos se encuentran en una lista enlazada, esta es recorrida para rellenar los buffers de carga utilizados por el algoritmo dentro de las FPGAs. La memoria no se encuentra de forma consecutiva, por lo que no es posible una carga directa, es necesario recorrer la lista para tratar cada dato de forma individual.

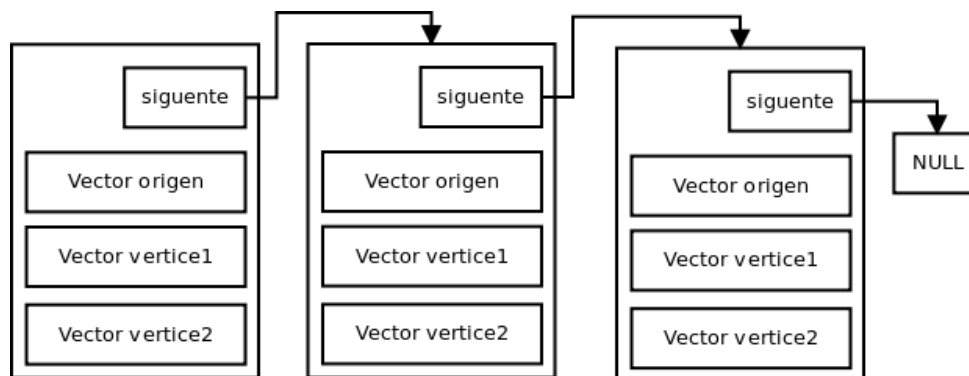


fig 10.2 triángulos en irradiación

Los triángulos están definidos por tres vectores que ubican sus tres vértices en el espacio, estos vectores son de tipo float, que en la implementación C para IA64 equivale a 4 bytes, o lo que es lo mismo 32 bits.

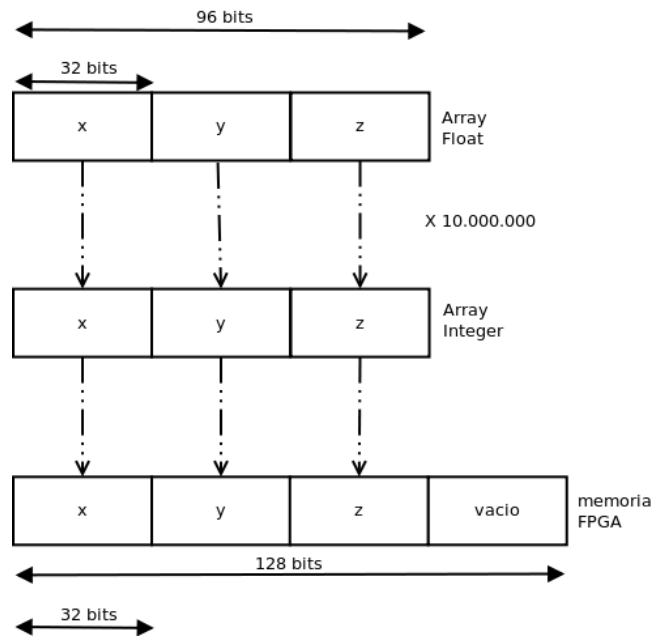


fig 10.3 conversión de datos

El diseño de nuestro algoritmo excluyó el uso de tipos decimales dentro de la FPGA, por lo que previamente se convierten a enteros, multiplicando por 10.000.000, y con ello haciendo un cambio de referencia en nuestras coordenadas que ha de ser desecho al leer los resultados. Una vez los datos son números enteros, cada uno de los vectores es copiado en una área de memoria distinta en la FPGA, de forma que simplifica las lecturas en el interior de esta. Lo que si hay que tener en cuenta es que las palabras dentro de la FPGA son de 16 bytes, por el modo de acceso a la memoria. En 16 bytes nos caben 4 interger de C, que tiene un ancho de 4 bytes en la implementación para IA64. Por lo tanto una cuarta parte de cada palabra, es ignorada, para simplificar la lógica de acceso a memoria .

12 FASE 10: PRUEBAS

El proyecto ha tenido un gran numero de pruebas durante su desarrollo, ya que no se disponía de la documentación necesaria para tener conocimiento sobre el comportamiento ante determinadas situaciones.

12.1 Pruebas de rendimiento Software

Las pruebas del software en paralelo se realizaron sobre el software, sin modificar, pero compilado expresamente para nuestra máquina, lo que implicaron ciertas modificaciones en los Makefiles.

Se renderizó una imagen de 4000 x 4000 píxeles con 7284 esferas. Todas las ejecuciones mostradas pueden tener cierta holgura en las medidas dependiendo de la disponibilidad de los recursos del sistema. Pero son fieles a las situaciones aquí explicadas.

```
linux> ./tachyon -res 4000 4000
../../scenes/balls.dat

Tachyon Parallel/Multiprocessor Ray Tracer
Version 0.98.7

Copyright 1994-2009,  John E. Stone
<john.stone@gmail.com>

-----

Scene Parsing Time:   0.0415 seconds
Scene contains 7384 objects.
Preprocessing Time:   0.0105 seconds
Rendering Progress:   100% complete
Ray Tracing Time:    117.9189 seconds
Image I/O Time:      0.9657 seconds
```

Ejecución en hilos vs MPI

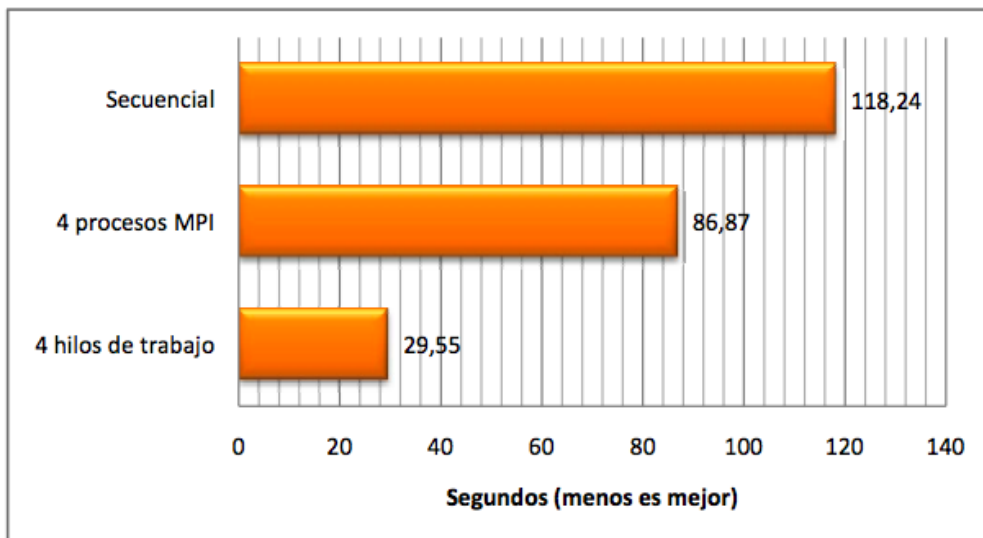


fig 12.1 comparación hilos y MPI

Una de las primeras pruebas a realizar, una vez hubimos escogido el software, fue comprobar el rendimiento con el uso de MPI y los hilos convencionales. El proyecto original contemplaba implementar todo el paralelismo software utilizando MPI, pero tras las pruebas del software original, se notó una considerable diferencia de rendimiento.

A la izquierda se puede ver el tiempo de ejecución con 4 hilos de trabajo, mientras que a la derecha se puede observar el tiempo de ejecución con MPI para 4 procesos.

<pre>linux-thr> ./tachyon -res 4000 4000 ../../scenes/balls.dat</pre> <p>Tachyon Parallel/Multiprocessor Ray Tracer Version 0.98.7</p> <p>Copyright 1994-2009, John E. Stone <john.stone@gmail.com></p> <p>-----</p> <p>Scene Parsing Time: 0.0425 seconds Scene contains 7384 objects. Preprocessing Time: 0.0112 seconds Rendering Progress: 100% complete Ray Tracing Time: 29.5523 seconds Image I/O Time: 1.0148 seconds</p>	<pre>linux-mpi> mpirun -np 4 ./tachyon -res 4000 4000 ../../scenes/balls.dat</pre> <p>Tachyon Parallel/Multiprocessor Ray Tracer Version 0.98.7</p> <p>Copyright 1994-2009, John E. Stone <john.stone@gmail.com></p> <p>-----</p> <p>Scene Parsing Time: 0.0478 seconds Scene contains 7384 objects. Preprocessing Time: 0.0241 seconds Rendering Progress: 100% complete Ray Tracing Time: 86.8760 seconds Image I/O Time: 2.4312 seconds</p>
--	---

Tras el análisis de los datos aquí mostrados se llegó a la siguiente conclusión. Durante la ejecución del test mediante MPI la copia de la memoria del proceso, para construir los procesos adicionales, la comunicación entre estos y la posterior recomposición de la salida, lastran el proceso de forma considerada.

Mientras que en el modelo de hilos, no es necesaria la copia de ningún conjunto de datos, las llamadas a funciones de diferentes hilos se introducen en diferentes colas y cada uno de ellos tiene sus datos locales, véase los rayos o datos intermedios, por lo que los accesos a memoria son directos, no hay copia de datos, ni envíos MPI que se encolan en el bus.

Ejecución iterativa vs recursiva

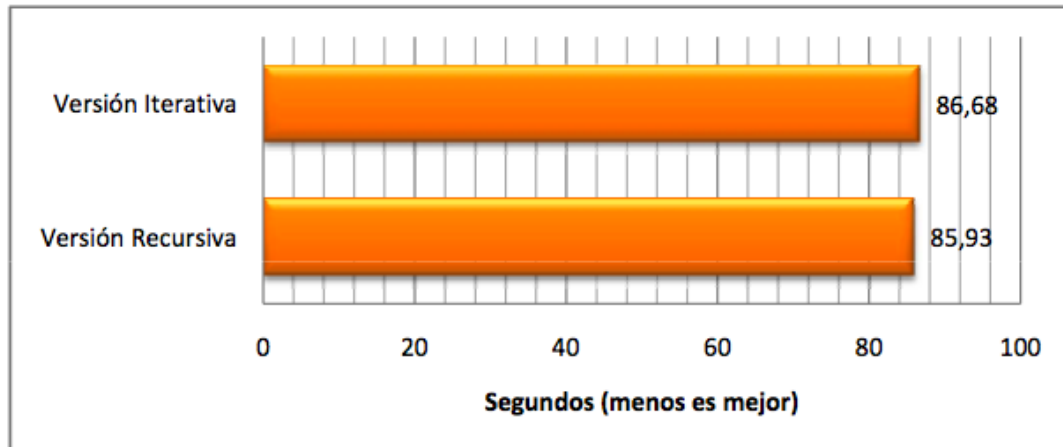


fig 12.2 versiones iterativa y recursiva

Para adecuar el código a nuestros intereses se modificaron las llamadas recursivas para que el código tuviera un comportamiento iterativo y disponer de los rayos en una cola.

<pre>Irradia-it> ./irradia -res 2000 2000 balls.dat</pre>	<pre>irradia> ./irradia -res 2000 2000 balls.dat</pre>
<p>IRRADIA IA64 Parallel Ray Tracer based in: Tachyon Parallel/Multiprocessor Ray Tracer Version 0.98.7</p> <p>Copyright 1994-2009, John E. Stone <john.stone@gmail.com></p> <p>-----</p> <p>Scene Parsing Time: 0.0528 seconds Scene contains 7384 objects. Preprocessing Time: 0.0223 seconds</p> <p>inicializando medidas para 1 procesadores comenzando, numero de nodos: 1 4 procesadores configurados 4 procesadores online Rendering Progress: 100% complete</p>	<p>IRRADIA IA64 Parallel Ray Tracer based in: Tachyon Parallel/Multiprocessor Ray Tracer Version 0.98.7</p> <p>Copyright 1994-2009, John E. Stone <john.stone@gmail.com></p> <p>-----</p> <p>Scene Parsing Time: 0.0502 seconds Scene contains 7384 objects. Preprocessing Time: 0.0224 seconds</p> <p>inicializando medidas para 1 procesadores comenzando, numero de nodos: 1 4 procesadores configurados 4 procesadores online Rendering Progress: 100% complete</p>

Ray Tracing Time: 86.6802 seconds Image I/O Time: 0.6279 seconds	Ray Tracing Time: 85.9384 seconds Image I/O Time: 0.6330 seconds
---	---

La sorpresa vino cuando se descubrió que el código tenía un comportamiento peor con la versión iterativa. Esto se debe a que, el manejo de la cola genera nuevas llamadas y saltos en el código que no existen con la llamada recursiva y debido a la optimización que realiza el compilador sobre la llamada a funciones la sobrecarga de los saltos y las llamadas adicionales no recompensan sobre la recursión original. A pesar de ser una reducción despreciable.

12.2 Pruebas de rendimiento de FPGA's y Rasclib

A fin de observar y comparar los tiempos de carga, reconfiguración y ejecución de nuestras FPGA's hemos optado por cronometrar los tiempos usando al algoritmo alg6, provisto en los ejemplos de RASC. Se muestran las salidas correspondientes a cada prueba. Se ha dejado la primera salida completa y se ha eliminado del resto más clara la lectura. En el CD adjunto a la memoria, en el directorio *Pruebas rendimiento alg6*, se pueden encontrar los códigos de los 4 ejemplos además de los resultados de ejecución.

Para conseguir medir los tiempos, hemos necesitado incluir un reloj de alta precisión en el código C, puesto que con el estándar *clock()* la precisión no era suficiente. En su lugar hemos usado la función *gettimeofday()*. *clock()* es una función dependiente del número de tics que una aplicación necesita para terminar, es dependiente del número de CPU's y tiene menor precisión que *gettimeofday()*, la cual esta implementa en el interior del kernel y nos devuelve unos tiempos de alta precisión.

Carga de ".bit", lectura de datos, ejecución y escritura de resultados (*alg6_tbase*).

En este ejemplo, se carga el ".bit" en la FPGA reconfigurándola, se leen datos del buffer de memoria, se ejecuta el algoritmo y se escribe en el citado buffer. El tiempo total son 71ms.

```
Reservado pool para 1 fpga/s, con nombre my_fpga

Configurado el algoritmo alg6 en la/s fpga/s 1 , en el pool my_fpga
La/s fpga/s ahora esta/n marcada/s en uso

Notificamos a rasclib que usaremos la cantidad de fpgas como un single device
para el algortimo: alg6
Modo RASCLIB_BUFFERED_IO

Indicamos que se moveran datos de memoria del host a a_in asociado en el
bitstream. Puntero de memoria host A

Indicamos que se moveran datos de memoria del host a b_in asociado en el
bitstream. Puntero de memoria host B

Indicamos que se moveran datos de memoria del host a c_in asociado en el
bitstream. Puntero de memoria host C
```

GO

Indicamos que se moveran datos de memoria del fpga desde d_out asociado en el bitstream a de memoria host D

Commit: se mandan todos los comandos encolados como una lista de comandos

Wait: bloquea hasta que todos los comandos han sido ejecutados

Close: libera todos los recursos del host asociados con el algoritmo, no libera fpgas

Tiempo de ejecución CON reconfiguracion: 70.922 milisegundos

Comparamos que D es lo mismo que SW_RES

CORRECTO FUNCIONA!

Return: Devuelve las fpgas al pool alg6

Release: Devuelve las fpgas al pool liberado my_fpga

Carga de ".bit", lectura de datos, ejecución, escritura de resultados, ejecución y escritura de resultados (alg6tx2).

En este ejemplo, se carga el “.bit” en la FPGA reconfigurándola, se leen datos del buffer de memoria, se ejecuta el algoritmo y se escribe en el citado buffer. Esto supone 72ms. Se vuelve a llamar a ejecución al algoritmo escribiendo de nuevo los datos en memoria consumiendo 5.1ms.

Carga de ".bit", lectura de datos, ejecución, escritura de resultados, lectura de datos, ejecución y escritura de resultados (alg6t).

En este ejemplo, se carga el “.bit” en la FPGA reconfigurándola, se leen datos del buffer de memoria, se ejecuta el algoritmo y se escribe en el citado buffer. Esto supone como siempre unos 72ms. Se leen de nuevo los datos y se vuelve a llamar a ejecución al algoritmo escribiendo de nuevo en memoria. Estas operaciones consumen 5.15ms.

Carga de ".bit", lectura de datos, ejecución, escritura de resultados, lectura de datos diferentes, ejecución y escritura de resultados (alg6_datos).

En este ejemplo, se carga el “.bit” en la FPGA reconfigurándola, se leen datos del buffer de memoria, se ejecuta el algoritmo y se escribe en el citado buffer. Esto supone como siempre unos 72ms. Se modifican los datos a leer por el dispositivo, se leen y se vuelve a llamar a ejecución al algoritmo escribiendo de nuevo en memoria. Estas operaciones consumen, al igual que en el caso anterior, 5.2ms.

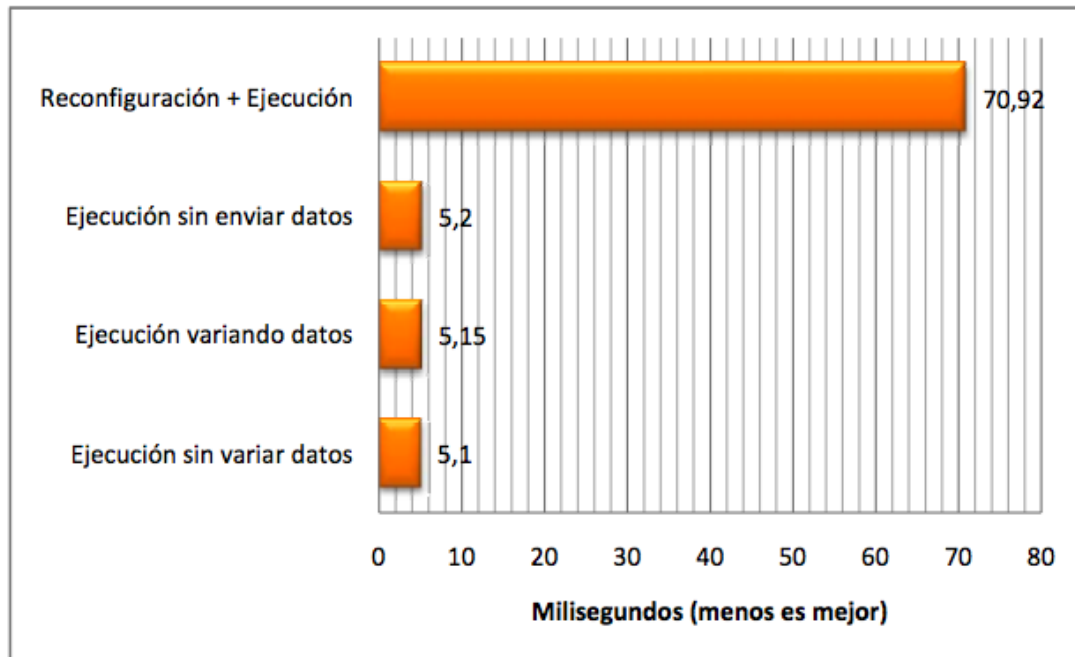


fig 12.3 pruebas reconfiguración

A la vista de los resultados de las pruebas anteriores podemos concluir:

- La ejecución sin reconfiguración consume solamente un 7% de tiempo necesario para reconfigurar y ejecutar, por tanto mantener y ejecutar el mismo algoritmo para muchos datos antes de modificarlo aumenta el rendimiento global de forma notoria.
- Si enviamos nuevos al buffer para ser leídos por la FPGA no se observa mejora temporal. Esto es debido a que la operación de lectura contenida en el código HDL sintetizado no ha sido modificada. Tampoco tendría sentido modificarla y reconfigurar el dispositivo a la vista de los resultados del punto anterior.
- Siempre que enviemos datos, consumimos el mismo tiempo sean o no idénticos a los que ya existiesen.

Debido a estos resultados, decidimos que en la parte computable mediante hardware, el algoritmo debería mantenerse el máximo tiempo en la FPGA junto con el mayor volumen de datos a procesar. Esta conclusión nos condujo a decidir que implementaríamos las intersecciones entre planos y rayos almacenando todos los planos en el buffer y variando solamente el rayo.

13 CONCLUSIONES, LOGROS Y LEGADO

Lamentablemente el tiempo del que se ha dispuesto, y los obstáculos que han surgido durante el transcurso del proyecto no nos han permitido alcanzar todas las metas dispuestas, pero se deja un legado de conocimiento que, sin duda, facilitará futuros desarrollos en esta máquina.

13.1 Conclusiones

Capacidad y rendimiento

La máquina dispone de 4 procesadores, y 4 dispositivos de hardware reconfigurable, por ello se puede realizar el siguiente cálculo teórico:

Velocidad de reloj máxima, 200MHz

Nuestro diseño tarda 3 ciclos en leer un triángulo de la memoria, 1 ciclo en procesarlo, y 1 ciclo en escribir el resultado en memoria, total 5 ciclos.

En procesar 4097 triángulos de la escena de prueba, el software tarda 10 segundos para un tamaño de 100 x 100 píxeles. Los tiempos de la ejecución tienen cierta holgura producida por la imprecisión de nuestros temporizadores.

```
irradia> ./irradia -res 100 100 ../tachyon/scenes/tetra.dat
IRRADIA IA64 Parallel Ray Tracer based in:
Tachyon Parallel/Multiprocessor Ray Tracer Version 0.98.7
Copyright 1994-2009, John E. Stone
<john.stone@gmail.com>

-----

Scene Parsing Time: 0.0398 seconds
Scene contains 4097 objects.
Preprocessing Time: 0.0000 seconds
comenzando, numero de nodos: 1
4 procesadores configurados
4 procesadores online
Rendering Progress: 100% complete

Ray Tracing Time: 10.8389 seconds
Iteration repeated: 11212 times
Triangles intersection time: 0.0010 seconds
Image I/O Time: 0.0248 seconds
```

Cada comprobación de intersección tarda 0,001 segundos de media, 1ms. Y se realizan 11212 veces para la configuración de la escena.

Para la misma ejecución en Hardware, en condiciones ideales con nuestro diseño se tardaría en computar las iteraciones para cada rayo una décima parte.

$$tiempo = \text{numero de triangulos} * \left(\frac{\text{numero ciclos}}{\text{frecuencia}} \right)$$

$$tiempo = 4097 * \left(\frac{5}{200 * 10^6} \right) = 0,000102425 \text{ segundos}$$

Con estos datos, reduciríamos la parte fundamental del calculo de la aplicación a una décima parte del tiempo que esta tomando en la configuración actual.

A esto, si añadimos que se disponen de 4 FPGAs, se puede añadir que el tiempo total para el cálculo de estas son una cuarta parte del tiempo computado aquí.

Coste de producción

La tecnología tiene una gran proyección teórica, pero ha demostrado tener unos desarrollos muy complicados, y más en fases tempranas, debido a la dificultad de encontrar información sobre el sistema.

La documentación disponible es escasa y resulta incómoda para hacer un diseño y para la producción en general. El hecho de que sea un desarrollo mixto, con software de bajo nivel y hardware asociado hace que, de por si, los proyectos sean complicados. Pero la falta de ejemplos y explicaciones agravan este hecho.

A la hora de diseñar el hardware, resulta complicado entender el sistema de memoria, ya que la única ayuda existente son los ejemplos en código Verilog, por lo definir la interfaz entre los dos entornos y el paso de datos es complicado.

Impresión sobre el producto

La impresión general sobre el producto es la siguiente

- Las máquinas RASC tienen unas capacidades de computo impresionantes.
- Tanto las librerías, la documentación y los ejemplos son escasos, parecen provenir de una versión temprana del producto y son síntoma de que es una tecnología inmadura.
- Los desarrollos son lentos e incómodos, debido al desconocimiento sobre el sistema, pero resultan satisfactorios e interesantes debido a la cantidad de tecnologías involucradas y al conocimiento que se adquiere con cada paso.

13.2 Logros

La mayoría de los objetivos iniciales del proyecto se han logrado. La adaptación de Irradia en el servidor Altix ha sido un logro y ha sido la primera vez que la máquina se ha puesto en funcionamiento con un proceso con una carga computacional considerable. De hecho con las herramientas dispuestas del proyecto ahora es posible renderizar de forma paralela entre los procesadores cualquier fichero de escena 3D de formato .ac3, formato muy común utilizado en los entornos 3D, y realizar el renderizado con tiempos inferiores a los ordenadores de uso común.

- Hemos analizado las distintas técnicas de procesamiento paralelo disponibles para la

aplicación y la máquina obteniendo unos resultados de test y pudiendo comparar los rendimientos de los mismos.

- Hemos logrado la conexión de las FPGAS con el servidor, realizado algunos algoritmos de pruebas, aunque lamentablemente el tiempo del que se ha dispuesto, y los obstáculos que han surgido durante el transcurso del proyecto no nos han permitido alcanzar todas las metas dispuestas, el algoritmo específico diseñado para las intersecciones de los triángulos no nos fue posible ejecutarlo en el servidor.
- La razón por la que no se pudo ejecutar el algoritmo es debido a las herramientas de configuración de las herramientas RASC y sus limitaciones o imposiciones, se explica más adelante con más detalle. Aun así hemos simulado el algoritmo que ejecutaríamos en las FPGAS mediante nuestra interfaz de aplicaciones para desarrolladores DummRasc y hemos comparado los rendimientos estimados teóricos que obtendríamos si nos hubiesen permitido la ejecución del algoritmo en las FPGAS y las estadísticas hablan de un 34,91% de mejora de rendimiento.
- Además hemos desarrollado dos librerías: DummyRasc que permite la simulación de cualquier algoritmo destinado a la FPGA por software y Rasconnect que permite la conexión con las FPGAS de manera más rápida y sencilla y se deja un legado de conocimiento para facilitar futuros desarrollos en la máquina.

Rendimiento logrado.

Se ha conseguido una velocidad de reloj máxima en la FPGA de 35MHz

Lo cual se traduce en un tiempo de cómputo de intersección de 0,000585286 Segundos

$$tiempo = 4097 * \left(\frac{5}{35 * 10^6} \right) = 0,000585286 \text{ segundos}$$

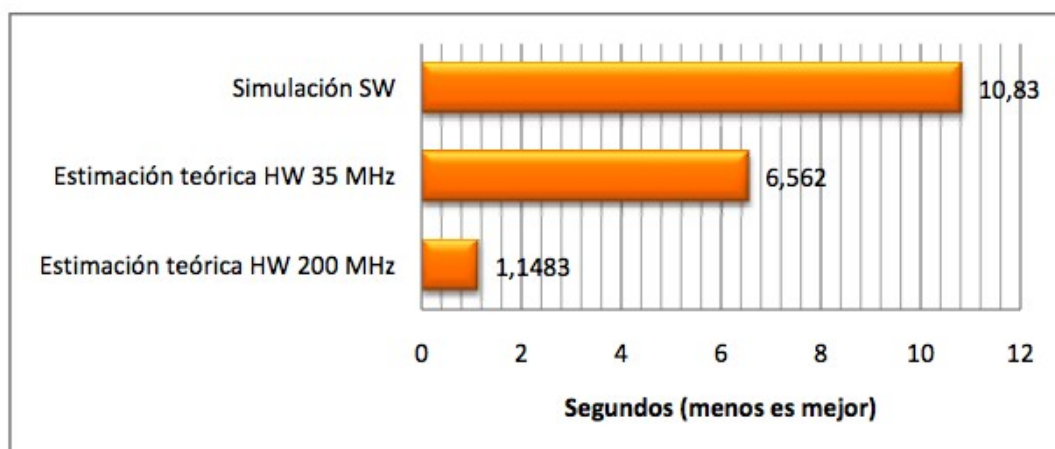


fig 13.1 comparación rendimiento

Lamentablemente estos datos solamente sirven como modelo teórico, ya que el sistema solo es capaz de proporcionar un tiempo de reloj mínimo a la FPGA de 50Mhz, y esta velocidad no permite una ejecución correcta de nuestro algoritmo.

13.3 Legado

Know how

La investigación, desarrollos y pruebas realizadas con el sistema han tenido como fruto un conocimiento sobre las características y capacidades de este, que en desarrollos futuros, representarán diseños más acertados desde el comienzo, reduciendo tiempos y mejorando la calidad de los productos finales.

Librerías y manuales

Durante el desarrollo, se encontraron dificultades y obstáculos que dieron lugar a subproductos durante su solución, la librería de pruebas DummyRASC es una prueba de ellas y esta diseñada para facilitar futuros proyectos que involucren el uso de esta tecnología.

Los manuales, y esta memoria, son una recopilación del conocimiento adquirido y fueron escritos con el objetivo de que puedan ser usados en trabajos futuros.

Diseños

Todos los diseños realizados, ya sean utilizados o no, sirven para ilustrar el funcionamiento de los dispositivos, la interfaz entre el software y el hardware y el modelo de memoria dentro del HW, por lo que tienen gran valor para diseños y modelos posteriores.

Trabajo futuro

Lamentablemente, el software desarrollado, y el hardware diseñado no alcanzan las expectativas de nuestro proyecto, por no haber obtenido un sistema funcional. Por lo que existe multitud de trabajo futuro.

- Rediseñar el hardware .
- Terminar las librerías de prueba.
- Revisar el Software, escoger un cambio de solución si es necesario.

14 PROBLEMAS Y SOLUCIONES

Durante el desarrollo del proyecto surgieron multitud de problemas, que nos acercaron a tener un conocimiento más real de las capacidades de la máquina pero que al mismo tiempo supusieron el motivo de modificar diseños y dar pasos atrás.

14.1 Problemas Software

La compatibilidad y el Makefile

La versión original de Tachyon viene con una lista limitada de compatibilidades y nuestra máquina no se encontraba disponible para la versión MPI. Tras investigar los Makefiles y realizar algunas modificaciones, se logró compilar sin ser necesario modificar mucho el código. Los problemas principales estaban relacionados con los temporizadores del sistema, que es una parte muy delicada ya que cada sistema tiene su propio grupo de temporizadores. Los hilos también eran delicados, ya que la primera versión se compiló en MacOS y los pthreads son mucho más cómodos, mientras que requieren opciones de compilación, añadir la directiva de precompilador y la librería para el enlazado.

La optimización en grids

Una vez fuimos capaces de compilar nuestros propios binarios para las FPGAs y teníamos claro el paso siguiente, así que se procedió a implementar el método que permitía cargar todos los objetos en la memoria de la FPGA, para ello se generó una nueva lista de objetos, que sería excluida del proceso normal, y sería ejecutado en las FPGAs.

El problema vino cuando, al implementar el cambio, se hizo notar una pérdida de rendimiento. Cuando se traza un rayo nuevo, se intersecta con todos los objetos, el cambio consiste en ejecutar la intersección de los triángulos recorriendo la lista de estos, para simular el comportamiento de la ejecución en la FPGA, este modo de trabajo resultaba muy lento.

Dicho comportamiento se debe a una optimización en la que no se había reparado hasta este momento dada la complejidad del código. Antes de ejecutar el algoritmo, los objetos son analizando y repartidos en celdas, que dividen el espacio, de forma que antes de procesar todos los objetos⁸ se comprueba si el rayo cruza o no el área determinada. Esta técnica es conocida como Grid hierarchy.

Nuestro código ejecuta la intersección para todos los triángulos en cada uno de los rayos, resultado en un trabajo mucho más pesado y lento.

14.2 Problemas Hardware

Los tipos de datos y el compute en las FPGAs

La programación C tiene unas características muy peculiares que permiten el tratamiento de la memoria y al mismo tiempo un sistema de tipos para hacer cómoda la programación. La integración con el código HDL requiere del acuerdo entre el ingeniero de hardware y el diseño del software, y en lo que se refiere a los tipos de datos, es necesario.

Tachyon hace un uso intensivo de los datos en forma decimal, todas las operaciones aritméticas se hacen en float, mientras que resulta muy incomodo hacer estas operaciones dentro de la FPGA, además de que se debe utilizar el tipo de datos compatible con el procesador del sistema.

⁸ En medium shade, por cada rayo se realiza la intersección con todos los objetos de la escena, mientras que full shade traza un rayo extra, por cada colisión del rayo anterior dirigido a las luces, para calcular las sombras reflejadas.

Por otra parte la alineación en memoria es un problema añadido al computo. RASC permite configurar sus FPGAs en dos opciones, con lo que se refiere al acceso a sus memorias, se pueden utilizar 3 streams, 2 de 128 bits y uno de 64 o 5 streams de 64 bits, nuestro desarrollo opto por utilizar los accesos de 128 bits a memoria, de forma que se puede leer un vector completo con cada ciclo de reloj.

Se diseño la interfaz de la siguiente manera, cada vector ocuparía 128 bits consecutivos en la memoria de las FPGAs, serían formateados de esa manera una vez se copian en esta. Pero también se realiza una modificación adicional, los valores en punto flotante se multiplican por 100.000.000 y se convierten a números enteros, consiguiendo con esto una gran precisión pero trabajando con números enteros. Los resultados son divididos una vez terminada la computación para continuar los cálculos en el tipo original, de esta forma nuestros vectores son formateados en 4 enteros con signo, cada uno de 4 bytes y el último de ellos sin uso.

Enlazando con el párrafo anterior, al usar vectores de datos de 32 bit's, la lógica necesaria era muy grande y no permitía usar los XtremeDSP's de la FPGA, generando un circuito capaz de funcionar a un máximo de 35Mhz. Dato proporcionado por Xilinx ISE. Como la herramienta de configuración proporcionada únicamente permitía rebajar la frecuencia del dispositivo hasta los 50Mhz no fue posible hacer funcionar el algoritmo en la FPGA.

14.3 Problemas varios

Devmgr y los errores con los ficheros .cfg

Devmgr es la interfaz con la cual se cargan y se gestionan los binarios destinados a ejecutarse en las FPGAs. Tras una carga fallida, devmgr mantenía un comportamiento errático y no se identificaba el porqué.

Se hizo reiniciar la máquina, sin conseguir ninguna mejora, finalmente Daniel Tabas, uno de los técnicos encargados de ésta, identifico el origen del problema, al hacer la carga devmgr copia tanto el binario como los ficheros de configuración de éste en una carpeta temporal, donde permanecen hasta que sera necesario cargarlo en una FPGA, la carga fallida había dejado los ficheros de configuración sin copiar, y devmgr era incapaz de encontrarlos o de recuperarse del error. Finalmente, se crearon dos ficheros vacíos de configuración y después de esto devmgr era capaz de eliminar el algoritmo fallido y de cargarlo correctamente.

Problemas al instalar XILINX en UNIX

Ya desde el principio nos encontramos con problemas instalando una herramienta que se especifica como necesaria pero que no se incluye en la máquina y que ha de instalarse en un ordenador externo a ella.

Descripción del error:

```
error while loading shared libraries: libstdc++.so.5: cannot open shared
object file: No such file or directory
```

Para solucionar el error son necesarias unas librerías de c++ para instalar la versión 9.2 que han sido reemplazadas por la version 6 en nuestra versión de Linux. Para solucionarlo es necesario descargarse las librerías antiguas mediante esta serie de comandos en la consola:

```
> cd /tmp
```

```
> wget http://security.ubuntu.com/ubuntu/pool/universe/i/ia32-libs/ia32libs\_2.7ubuntu6.1\_amd64.deb

> dpkg-deb -x ia32-libs_2.7ubuntu6.1_amd64.deb ia32-libs

> sudo cp ia32-libs/usr/lib32/libstdc++.so.5.0.7 /usr/lib/

> cd /usr/lib

> sudo ln -s libstdc++.so.5.0.7 libstdc++.so.5
```

14.4 Conclusión de los Problemas

Es comprensible que durante la ejecución de un determinado software ocurran problemas o errores. Lo que no nos parece racional es que éstos problemas no se encuentren documentados por parte de los desarrolladores de la herramienta concreta. Durante la Fase 7 nos encontramos con muchos obstáculos, y por eso creemos conveniente documentarlos.

Gracias a la ayuda de la Universidad Politécnica de Ciudad Real, que también posee un servidor de la misma marca pudimos continuar con el desarrollo del proyecto y la posterior finalización del mismo. Tuvimos que recurrir a terceros para corregir algunos de los problemas que tienen raíz en las herramientas diseñadas específicamente para el servidor, lo cual nos ocasionó una ralentización en el transcurso del proyecto, concretamente en esta fase.

15 REFERENCIAS Y BIBLIOGRAFÍA

Bibliografía

- RASC Reference Manual
Manual de referencia de SGI, disponible en internet e incluido en el CD
<http://techpubs.sgi.com/library/tpl/cgi-bin/download.cgi?coll=linux&db=bks&docnumber=007-4718-007>
- Manual de referencia VHDL
Manual de referencia para la programación en el lenguaje VHDL
http://www.usna.edu/EE/ee462/MANUALS/vhdl_ref.pdf

Referencias de interés

- Proyecto con similar enfoque,
<http://upcommons.upc.edu/pfc/handle/2099.1/9074>
- Opciones barajadas, programas para render con raytracing
<http://www.povray.org/>
<http://radsite.lbl.gov/radiance/HOME.html>
<http://www-graphics.stanford.edu/~cek/rayshade/rayshade.html>
<http://jedi.ks.uiuc.edu/~johns/raytracer/>
- Comparativas e información de interés sobre FPGAs
http://www.linleygroup.com/Reports/fpga_guide.html
<http://blog.linleygroup.com/2010/03/innovation-disrupts-fpga-duopoly.htm>
<http://www.dspdesignline.com/news/218501184;jsessionid=VMSYCQS2DWHHHQE1GHPCKH4ATMY32JVN?pgno=2>

16. CONTENIDO DEL CD

Este proyecto adjunta un cd con contenido de interés para futuros desarrollos. A continuación se detalla la estructura de carpetas.

- Apendices
 - Apendice Diagramas
 - Apendice DummyRASC
- imágenes
 - Imágenes renderizadas
 - Escenas
 - Escenas de Tachyon
- manuales
 - Manual de referencia RASC
 - Manual de programación VHDL
- pruebas
 - Pruebas rendimiento alg6
 - algoritmo 6 con registros
 - prueba de copia de buffers
- software
 - dRaslib
 - irradia
 - vhdl

Los alumnos abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Ignacio Arroyo Gómez

Luis Fernando Ayuso Pérez

Pablo Escobar de la Oliva

